CHAPTER 7

Linux Neighbouring Subsystem

This chapter discusses the Linux neighbouring subsystem and its implementation in Linux. The neighbouring subsystem is responsible for the discovery of the presence of nodes on the same link and for translation of L3 (network layer) addresses to L2 (link layer) addresses. L2 addresses are needed to build the L2 header for outgoing packets, as described in the next section. The protocol that implements this translation is called the Address Resolution Protocol (ARP) in IPv4 and Neighbour Discovery protocol (NDISC or ND) in IPv6. The neighbouring subsystem provides a protocol-independent infrastructure for performing L3-to-L2 mappings. The discussion in this chapter, however, is restricted to the most common cases—namely, the neighbouring subsystem usage in IPv4 and in IPv6. Keep in mind that the ARP protocol, like the ICMP protocol discussed in Chapter 3, is subject to security threats—such as ARP poisoning attacks and ARP spoofing attacks (security aspects of the ARP protocol are beyond the scope of this book).

I first discuss the common neighbouring data structures in this chapter and some important API methods, which are used both in IPv4 and in IPv6. Then I discuss the particular implementations of the ARP protocol and NDISC protocol. You will see how a neighbour is created and how it is freed, and you will learn about the interaction between userspace and the neighbouring subsystem. You will also learn about ARP requests and ARP replies, about NDISC neighbour solicitation and NDISC neighbour advertisements, and about a mechanism called Duplicate Address Detection (DAD), which is used by the NDISC protocol to avoid duplicate IPv6 addresses.

The Neighbouring Subsystem Core

What is the neighbouring subsystem needed for? When a packet is sent over the L2 layer, the L2 destination address is needed to build an L2 header. Using the neighbouring subsystem solicitation requests and solicitation replies, the L2 address of a host can be found out given its L3 address (or the fact that such L3 address does not exist). In Ethernet, which is the most commonly used link layer (L2), the L2 address of a host is its MAC address. In IPv4, ARP is the neighbouring protocol, and solicitation requests and solicitation replies are called ARP requests and ARP replies, respectively. In IPv6, the neighbouring protocol is NDISC, and solicitation requests and solicitation replies are called neighbour solicitations and neighbour advertisements, respectively.

There are cases where the destination address can be found without any help from the neighbouring subsystem—for example, when a broadcast is sent. In this case, the destination L2 address is fixed (for example, it is FF:FF:FF:FF:FF:FF in Ethernet). Or when the destination address is a multicast address, there is a fixed mapping between the L3 multicast address to its L2 address. I discuss such cases in the course of this chapter.

The basic data structure of the Linux neighbouring subsystem is the neighbour. A *neighbour* represents a network node that is attached to the same link (L2). It is represented by the `neighbour` structure. This representation is not unique for a particular protocol. However, as mentioned, the discussion of the `neighbour` structure will be restricted to its use in the IPv4 and in the IPv6 protocols. Let's take a look in the `neighbour` structure:

```
struct neighbour {
        struct neighbour __rcu  *next;
        struct neigh_table      *tbl;
```

```
        struct neigh_parms      *parms;
        unsigned long           confirmed;
        unsigned long           updated;
        rwlock_t                lock;
        atomic_t                refcnt;
        struct sk_buff_head     arp_queue;
        unsigned int            arp_queue_len_bytes;
        struct timer_list       timer;
        unsigned long           used;
        atomic_t                probes;
        __u8                    flags;
        __u8                    nud_state;
        __u8                    type;
        __u8                    dead;
        seqlock_t               ha_lock;
        unsigned char           ha[ALIGN(MAX_ADDR_LEN, sizeof(unsigned long))];
        struct hh_cache         hh;
        int                     (*output)(struct neighbour *, struct sk_buff *);
        const struct neigh_ops  *ops;
        struct rcu_head         rcu;
        struct net_device       *dev;
        u8                      primary_key[0];
};
```

(include/net/neighbour.h)

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

- `next`: A pointer to the next neighbour on the same bucket in the hash table.
- `tbl`: The neighbouring table associated to this neighbour.
- `parms`: The `neigh_parms` object associated to this `neighbour`. It is initialized by the `constructor` method of the associated neighbouring table. For example, in IPv4 the `arp_constructor()` method initializes `parms` to be the `arp_parms` of the associated network device. Do not confuse it with the `neigh_parms` object of the neighbouring table.
- `confirmed`: Confirmation timestamp (discussed later in this chapter).
- `refcnt`: Reference counter. Incremented by the `neigh_hold()` macro and decremented by the `neigh_release()` method. The `neigh_release()` method frees the neighbour object by calling the `neigh_destroy()` method only if after decrementing the reference counter its value is 0.
- `arp_queue`: A queue of unresolved SKBs. Despite the name, this member is not unique to ARP and is used by other protocols, such as the NDISC protocol.
- `timer`: Every `neighbour` object has a timer; the timer callback is the `neigh_timer_handler()` method. The `neigh_timer_handler()` method can change the Network Unreachability Detection (NUD) state of the neighbour. When sending solicitation requests, and the state of the neighbour is NUD_INCOMPLETE or NUD_PROBE, and the number of solicitation requests probes is higher or equal to `neigh_max_probes()`, then the state of the neighbour is set to be NUD_FAILED, and the `neigh_invalidate()` method is invoked.
- `ha_lock`: Provides access protection to the neighbour hardware address (`ha`).

- ha: The hardware address of the neighbour object; in the case of Ethernet, it is the MAC address of the neighbour.
- hh: A hardware header cache of the L2 header (An hh_cache object).
- output: A pointer to a transmit method, like the neigh_resolve_output() method or the neigh_direct_output() method. It is dependent on the NUD state and as a result can be assigned to different methods during a neighbour lifetime. When initializing the neighbour object in the neigh_alloc() method, it is set to be the neigh_blackhole() method, which discards the packet and returns -ENETDOWN.

  And here are the helper methods (methods which set the output callback):

  - void neigh_connect(struct neighbour *neigh)

    Sets the output() method of the specified neighbour to be neigh->ops->connected_output.

  - void neigh_suspect(struct neighbour *neigh)

    Sets the output() method of the specified neighbour to be neigh->ops->output.

- nud_state: The NUD state of the neighbour. The nud_state value can be changed dynamically during the lifetime of a neighbour object. Table 7-1 in the "Quick Reference" section at the end of this chapter describes the basic NUD states and their Linux symbols. The NUD state machine is very complex; I do not delve into all of its nuances in this book.
- dead: A flag that is set when the neighbour object is alive. It is initialized to 0 when creating a neighbour object, at the end of the __neigh_create() method. The neigh_destroy() method will fail for neighbour objects whose dead flag is not set. The neigh_flush_dev() method sets the dead flag to 1 but does not yet remove the neighbour entry. The removal of neighbours marked as dead (their dead flag is set) is done later, by the garbage collectors.
- primary_key: The IP address (L3) of the neighbour. A lookup in the neighbouring tables is done with the primary_key. The primary_key length is based on which protocol is used. For IPv4, for example, it should be 4 bytes. For IPv6 it should be sizeof(struct in6_addr), as the in6_addr structure represents an IPv6 address. Therefore, the primary_key is defined as an array of 0 bytes, and when allocating a neighbour it should be taken into account which protocol is used. See the explanation about entry_size and key_len later in this chapter, in the description of the neigh_table structure members.

To avoid sending solicitation requests for each new packet that is transmitted, the kernel keeps the mapping between L3 addresses and L2 addresses in a data structure called a neighbouring table; in the case of IPv4, it is the ARP table (sometimes also called the ARP cache, though they are the same)—in contrast to what you saw in the IPv4 routing subsystem in Chapter 5: the routing cache, before it was removed, and the routing table, were two different entities, which were represented by two different data structures. In the case of IPv6, the neighbouring table is the NDISC table (also known as the NDISC cache). Both the ARP table (arp_tbl) and the NDISC table (nd_tbl) are instances of the neigh_table structure. Let's take a look at the neigh_table structure:

```
struct neigh_table {
        struct neigh_table      *next;
        int                     family;
        int                     entry_size;
        int                     key_len;
        __u32                   (*hash)(const void *pkey,
                                        const struct net_device *dev,
                                        __u32 *hash_rnd);
```

```
        int                     (*constructor)(struct neighbour *);
        int                     (*pconstructor)(struct pneigh_entry *);
        void                    (*pdestructor)(struct pneigh_entry *);
        void                    (*proxy_redo)(struct sk_buff *skb);
        char                    *id;
        struct neigh_parms      parms;
        /* HACK. gc_* should follow parms without a gap! */
        int                     gc_interval;
        int                     gc_thresh1;
        int                     gc_thresh2;
        int                     gc_thresh3;
        unsigned long           last_flush;
        struct delayed_work     gc_work;
        struct timer_list       proxy_timer;
        struct sk_buff_head     proxy_queue;
        atomic_t                entries;
        rwlock_t                lock;
        unsigned long           last_rand;
        struct neigh_statistics __percpu *stats;
        struct neigh_hash_table __rcu *nht;
        struct pneigh_entry     **phash_buckets;
};
```

(include/net/neighbour.h)

Here are some important members of the neigh_table structure:

- next: Each protocol creates its own neigh_table instance. There is a linked list of all the neighbouring tables in the system. The neigh_tables global variable is a pointer to the beginning of the list. The next variable points to the next item in this list.
- family: The protocol family: AF_INET for the IPv4 neighbouring table (arp_tbl), and AF_INET6 for the IPv6 neighbouring table (nd_tbl).
- entry_size: When allocating a neighbour entry by the neigh_alloc() method, the size for allocation is tbl->entry_size + dev->neigh_priv_len. Usually the neigh_priv_len value is 0. Before kernel 3.3, the entry_size was explicitly initialized to be sizeof(struct neighbour) + 4 for ARP, and sizeof(struct neighbour) + sizeof(struct in6_addr) for NDISC. The reason for this initialization was that when allocating a neighbour, you want to allocate space also for the primary_key[0] member. From kernel 3.3, the enrty_size was removed from the static initialization of arp_tbl and ndisc_tbl, and the entry_size initialization is done based on the key_len in the core neighbouring layer, by the neigh_table_init_no_netlink() method.
- key_len: The size of the lookup key; it is 4 bytes for IPv4, because the length of IPv4 address is 4 bytes, and it is sizeof(struct in6_addr) for IPv6. The in6_addr structure represents an IPv6 address.
- hash: The hash function for mapping a key (L3 address) to a specific hash value; for ARP it is the arp_hash() method. For NDISC it is the ndisc_hash() method.
- constructor: This method performs protocol-specific initialization when creating a neighbour object. For example, arp_constructor() for ARP in IPv4 and ndisc_constructor() for NDISC in IPv6. The constructor callback is invoked by the __neigh_create() method. It returns 0 on success.

- pconstructor: A method for creation of a neighbour proxy entry; it is not used by ARP, and it is pndisc_constructor for NDISC. This method should return 0 upon success. The pconstructor method is invoked from the pneigh_lookup() method if the lookup fails, on the condition that the pneigh_lookup() was invoked with creat = 1.
- pdestructor: A method for destroying a neighbour proxy entry. Like the pconstructor callback, the pdestructor is not used by ARP, and it is pndisc_destructor for NDISC. The pdestructor method is invoked from the pneigh_delete() method and from the pneigh_ifdown() method.
- id: The name of the table; it is arp_cache for IPv4 and ndisc_cache for IPv6.
- parms: A neigh_parms object: each neighbouring table has an associated neigh_parms object, which consists of various configuration settings, like reachability information, various timeouts, and more. The neigh_parms initialization is different in the ARP table and in the NDISC table.
- gc_interval: Not used directly by the neighbouring core.
- gc_thresh1, gc_thresh2, gc_thresh3: Thresholds of the number of neighbouring table entries. Used as criteria to activation of the synchronous garbage collector (neigh_forced_gc) and in the neigh_periodic_work() asynchronous garbage collector handler. See the explanation about allocating a neighbour object in the "Creating and Freeing a Neighbour" section later in this chapter. In the ARP table, the default values are: gc_thresh1 is 128, gc_thresh2 is 512, and gc_thresh3 is 1024. These values can be set by procfs. The same default values are also used in the NDISC table in IPv6. The IPv4 procfs entries are:
  - /proc/sys/net/ipv4/neigh/default/gc_thresh1
  - /proc/sys/net/ipv4/neigh/default/gc_thresh2
  - /proc/sys/net/ipv4/neigh/default/gc_thresh3

  and for IPv6, these are the procfs entries:
  - /proc/sys/net/ipv6/neigh/default/gc_thresh1
  - /proc/sys/net/ipv6/neigh/default/gc_thresh2
  - /proc/sys/net/ipv6/neigh/default/gc_thresh3
- last_flush: The most recent time when the neigh_forced_gc() method ran. It is initialized to be the current time (jiffies) in the neigh_table_init_no_netlink () method.
- gc_work: Asynchronous garbage collector handler. Set to be the neigh_periodic_work() timer by the neigh_table_init_no_netlink() method. The delayed_work struct is a type of a work queue. Before kernel 2.6.32, the neigh_periodic_timer() method was the asynchronous garbage collector handler; it processed only one bucket and not the entire neighbouring hash table. The neigh_periodic_work() method first checks whether the number of the entries in the table is less than gc_thresh1, and if so, it exits without doing anything; then it recomputes the reachable time (the reachable_time field of parms, which is the neigh_parms object associated with the neighbouring table). Then it scans the neighbouring hash table and removes entries which their state is not NUD_PERMANENT or NUD_IN_TIMER, and which their reference count is 1, and if one of these conditions is met: either they are in the NUD_FAILED state or the current time is after their used timestamp + gc_staletime (gc_staletime is a member of the neighbour parms object). Removal of the neighbour entry is done by setting the dead flag to 1 and calling the neigh_cleanup_and_release() method.

- proxy_timer: When a host is configured as an ARP proxy, it is possible to avoid immediate processing of solicitation requests and to process them with some delay. This is due to the fact that for an ARP proxy host, there can be a large number of solicitation requests (as opposed to the case when the host is not an ARP proxy, when you usually have a small amount of ARP requests). Sometimes you may prefer to delay the reply to such broadcasts so that you can give priority to hosts that own such IP addresses to be the first to get the request. This delay is a random value up to the proxy_delay parameter. The ARP proxy timer handler is the neigh_proxy_process() method. The proxy_timer is initialized by the neigh_table_init_no_netlink() method.
- proxy_queue: Proxy ARP queue of SKBs. SKBs are added with the pneigh_enqueue() method.
- stats: The neighbour statistics (neigh_statistics) object; consists of per CPU counters like allocs, which is the number of neighbour objects allocated by the neigh_alloc() method, or destroys, which is the number of neighbour objects which were freed by the neigh_destroy() method, and more. The neighbour statistics counters are incremented by the NEIGH_CACHE_STAT_INC macro. Note that because the statistics are per CPU counters, the macro this_cpu_inc() is used by this macro. You can display the ARP statistics and the NDISC statistics with cat /proc/net/stat/arp_cache and cat/proc/net/stat/ndisc_cache, respectively. In the "Quick Reference" section at the end of this chapter, there is a description of the neigh_statistics structure, specifying in which method each counter is incremented.
- nht: The neighbour hash table (neigh_hash_table object).
- phash_buckets: The neighbouring proxy hash table; allocated in the neigh_table_init_no_netlink() method.

The initialization of the neighbouring table is done with the neigh_table_init() method:

- In IPv4, the ARP module defines the ARP table (an instance of the neigh_table structure named arp_tbl) and passes it as an argument to the neigh_table_init() method (see the arp_init() method in net/ipv4/arp.c).
- In IPv6, the NDISC module defines the NDSIC table (which is also an instance of the neigh_table structure named nd_tbl) and passes it as an argument to the neigh_table_init() method (see the ndisc_init() method in net/ipv6/ndisc.c).

The neigh_table_init() method also creates the neighbouring hash table (the nht object) by calling the neigh_hash_alloc() method in the neigh_table_init_no_netlink() method, allocating space for eight hash entries:

```
static void neigh_table_init_no_netlink(struct neigh_table *tbl)
{
    . . .
    RCU_INIT_POINTER(tbl->nht, neigh_hash_alloc(3));
    . . .
}

static struct neigh_hash_table *neigh_hash_alloc(unsigned int shift)
{
```

The size of the hash table is 1<< shift (when size <= PAGE_SIZE):

```
    size_t size = (1 << shift) * sizeof(struct neighbour *);
    struct neigh_hash_table *ret;
    struct neighbour __rcu **buckets;
    int i;
```

```
    ret = kmalloc(sizeof(*ret), GFP_ATOMIC);
    if (!ret)
        return NULL;
    if (size <= PAGE_SIZE)
        buckets = kzalloc(size, GFP_ATOMIC);
    else
        buckets = (struct neighbour __rcu **)
              __get_free_pages(GFP_ATOMIC | __GFP_ZERO,
                       get_order(size));
    . . .

}
```

You may wonder why you need the `neigh_table_init_no_netlink()` method—why not perform all of the initialization in the `neigh_table_init()` method? The `neigh_table_init_no_netlink()` method performs all of the initializations of the neighbouring tables, except for linking it to the global linked list of neighbouring tables, `neigh_tables`. Originally such initialization, without linking to the `neigh_tables` linked list, was needed for ATM, and as a result the `neigh_table_init()` method was split, and the ATM clip module called the `neigh_table_init_no_netlink()` method instead of calling the `neigh_table_init()` method; however, over time, a different solution was found in ATM. Though the ATM clip module does not invoke the `neigh_table_init_no_netlink()` method anymore, the split of these methods remained, perhaps in case it is needed in the future.

I should mention that each L3 protocol that uses the neighbouring subsystem also registers a protocol handler: for IPv4, the handler for ARP packets (packets whose type in their Ethernet header is 0x0806) is the `arp_rcv()` method:

```
static struct packet_type arp_packet_type __read_mostly = {
         .type = cpu_to_be16(ETH_P_ARP),
         .func = arp_rcv,
 };

 void __init arp_init(void)
 {
     . . .
         dev_add_pack(&arp_packet_type);
     . . .
}
```

(net/ipv4/arp.c)

For IPv6, the neighbouring messages are ICMPv6 messages, so they are handled by the `icmpv6_rcv()` method, which is the ICMPv6 handler. There are five ICMPv6 neighbouring messages; when each of them is received (by the `icmpv6_rcv()` method), the `ndisc_rcv()` method is invoked to handle them (see `net/ipv6/icmp.c`). The `ndisc_rcv()` method is discussed in a later section in this chapter. Each neighbour object defines a set of methods by the `neigh_ops` structure. This is done by its `constructor` method. The `neigh_ops` structure contains a protocol family member and four function pointers:

```
struct neigh_ops {
        int      family;
        void     (*solicit)(struct neighbour *, struct sk_buff *);
        void     (*error_report)(struct neighbour *, struct sk_buff *);
        int      (*output)(struct neighbour *, struct sk_buff *);
        int      (*connected_output)(struct neighbour *, struct sk_buff *);
};
```

(include/net/neighbour.h)

- `family`: AF_INET for IPv4 and AF_INET6 for IPv6.
- `solicit`: This method is responsible for sending the neighbour solicitation requests: in ARP it is the `arp_solicit()` method, and in NDISC it is the `ndisc_solicit()` method.
- `error_report`: This method is called from the `neigh_invalidate()` method when the neighbour state is NUD_FAILED. This happens, for example, after some timeout when a solicitation request is not replied.
- `output`: When the L3 address of the next hop is known, but the L2 address is not resolved, the `output` callback should be `neigh_resolve_output()`.
- `connected_output`: The output method of the neighbour is set to be `connected_output()` when the neighbour state is NUD_REACHABLE or NUD_CONNECTED. See the invocations of `neigh_connect()` in the `neigh_update()` method and in the `neigh_timer_handler()` method.

Creating and Freeing a Neighbour

A neighbour is created by the `__neigh_create()` method:

```
struct neighbour *__neigh_create(struct neigh_table *tbl, const void *pkey, struct
net_device *dev, bool want_ref)
```

First, the `__neigh_create()` method allocates a neighbour object by calling the `neigh_alloc()` method, which also performs various initializations. There are cases when the `neigh_alloc()` method calls the synchronous garbage collector (which is the `neigh_forced_gc()` method):

```
static struct neighbour *neigh_alloc(struct neigh_table *tbl, struct net_device *dev)
{
        struct neighbour *n = NULL;
        unsigned long now = jiffies;
        int entries;

        entries = atomic_inc_return(&tbl->entries) - 1;
```

If the number of table entries is greater than `gc_thresh3` (1024 by default) or if the number of table entries is greater than `gc_thresh2` (512 by default), and the time passed since the last flush is more than 5 Hz, the synchronous garbage collector method is invoked (the `neigh_forced_gc()` method). If after running the `neigh_forced_gc()` method, the number of table entries is greater than `gc_thresh3` (1024), you do not allocate a neighbour object and return NULL:

```
        if (entries >= tbl->gc_thresh3 ||
            (entries >= tbl->gc_thresh2 &&
            time_after(now, tbl->last_flush + 5 * HZ))) {
               if (!neigh_forced_gc(tbl) &&
                   entries >= tbl->gc_thresh3)
                       goto out_entries;
        }
```

Then the `__neigh_create()` method performs the protocol-specific setup by calling the `constructor` method of the specified neighbouring table (`arp_constructor()` for ARP, `ndisc_constructor()` for NDISC). In the constructor method, special cases like multicast or loopback addresses are handled. In the `arp_constructor()` method, for example, you call the `arp_mc_map()` method to set the hardware address of the neighbour (`ha`) according to the neighbour IPv4 `primary_key` address, and you set the `nud_state` to be NUD_NOARP, because multicast addresses don't need ARP. In the `ndisc_constructor()` method, for example, you do something quite similar when handling multicast addresses: you call the `ndisc_mc_map()` to set the hardware address of the neighbour (`ha`) according to the neighbour IPv6 `primary_key` address, and you again set the `nud_state` to be NUD_NOARP. There's also special treatment for broadcast addresses: in the `arp_constructor()` method, for example, when the neighbour type is RTN_BROADCAST, you set the neighbour hardware address (`ha`) to be the network device broadcast address (the `broadcast` field of the `net_device` object), and you set the `nud_state` to be NUD_NOARP. Note that the IPv6 protocol does not implement traditional IP broadcast, so the notion of a broadcast address is irrelevant (there is a link-local all nodes multicast group at address `ff02::1`, though). There are two special cases when additional setup needs to be done:

- When the `ndo_neigh_construct()` callback of the `netdev_ops` is defined, it is invoked. In fact, this is done only in the classical IP over ATM code (`clip`); see `net/atm/clip.c`.
- When the `neigh_setup()` callback of the `neigh_parms` object is defined, it is invoked. This is used, for example, in the bonding driver; see `drivers/net/bonding/bond_main.c`.

When trying to create a `neighbour` object by the `__neigh_create()` method, and the number of the neighbour entries exceeds the hash table size, it must be enlarged. This is done by calling the `neigh_hash_grow()` method, like this:

```
struct neighbour *__neigh_create(struct neigh_table *tbl, const void *pkey,
                struct net_device *dev, bool want_ref)
{
     . . .
```

The hash table size is `1 << nht->hash_shift`; the hash table must be enlarged if it is exceeded:

```
     if (atomic_read(&tbl->entries) > (1 << nht->hash_shift))
        nht = neigh_hash_grow(tbl, nht->hash_shift + 1);
     . . .
}
```

When the `want_ref` parameter is true, you will increment the neighbour reference count within this method. You also initialize the `confirmed` field of the `neighbour` object:

```
n->confirmed = jiffies - (n->parms->base_reachable_time << 1);
```

It is initialized to be a little less than the current time, `jiffies` (for the simple reason that you want reachability confirmation to be required sooner). At the end of the `__neigh_create()` method, the `dead` flag is initialized to be 0, and the `neighbour` object is added to the neighbour hash table.

The `neigh_release()` method decrements the reference counter of the neighbour and frees it when it reaches zero by calling the `neigh_destroy()` method. The `neigh_destroy()` method will verify that the neighbour is marked as `dead`: neighbours whose `dead` flag is 0 will not be removed.

In this section, you learned about the kernel methods to create and free a neighbour. Next you will learn how adding and deleting a neighbour entry can be triggered from userspace, as well as how to display the neighbouring table, with the `arp` command for IPv4 and the `ip` command for IPv4/IPv6.

Interaction Between Userspace and the Neighbouring Subsystem

Management of the ARP table is done with the `ip neigh` command of the `iproute2` package or with the `arp` command of the `net-tools` package. Thus, you can display the ARP table by running, from the command line, one of the following commands:

- `arp`: Handled by the `arp_seq_show()` method in `net/ipv4/arp.c`.
- `ip neigh show` (or `ip neighbour show`): Handled by the `neigh_dump_info()` method in `net/core/neighbour.c`.

Note that the `ip neigh show` command shows the NUD states of the neighbouring table entries (like NUD_REACHABLE or NUD_STALE). Note also that the `arp` command can display only the IPv4 neighbouring table (the ARP table), whereas with the `ip` command you can display both the IPv4 ARP table and the IPv6 neighbouring table. If you want to display only the IPv6 neighbouring table, you should run `ip -6 neigh show`.

The ARP and NDISC modules also export data via `procfs`. That means you can display the ARP table by running `cat /proc/net/arp` (this `procfs` entry is handled by the `arp_seq_show()` method, which is the same method that handles the `arp` command, as mentioned earlier). Or you can display ARP statistics by `cat /proc/net/stat/arp_cache`, and you can display the NDISC statistics by `cat /proc/net/stat/ndisc_cache` (both are handled by the `neigh_stat_seq_show()` method).

You can add an entry with `ip neigh add`, which is handled by the `neigh_add()` method. When running `ip neigh add`, you can specify the state of the entry which you are adding (like NUD_PERMANENT, NUD_STALE, NUD_REACHABLE and so on). For example:

```
ip neigh add 192.168.0.121 dev eth0 lladdr 00:30:48:5b:cc:45 nud permanent
```

Deleting an entry can be done by `ip neigh del`, and is handled by the `neigh_delete()` method. For example:

```
ip neigh del 192.168.0.121 dev eth0
```

Adding an entry to the proxy ARP table can be done with `ip neigh add proxy`. For example:

```
ip neigh add proxy 192.168.2.11 dev eth0
```

The addition is handled again by the `neigh_add()` method. In this case, the NTF_PROXY flag is set in the data passed from userspace (see the `ndm_flags` field of the `ndm` object), and therefore the `pneigh_lookup()` method is called to perform a lookup in the proxy neighbouring hash table (`phash_buckets`). In case the lookup failed, the `pneigh_lookup()` method adds an entry to the proxy neighbouring hash table.

Deleting an entry from the proxy ARP table can be done with `ip neigh del proxy`. For example:

```
ip neigh del proxy 192.168.2.11 dev eth0
```

The deletion is handled by the `neigh_delete()` method. Again, in this case the NTF_PROXY flag is set in the data passed from userspace (see the `ndm_flags` field of the `ndm` object), and therefore the `pneigh_delete()` method is called to delete the entry from the proxy neighbouring table.

With the `ip ntable` command, you can control the parameters for the neighbouring tables. For example:

- `ip ntable show`: Shows the parameters for all the neighbouring tables.
- `ip ntable change`: Change a value of a parameter of a neighbouring table. Handled by the `neightbl_set()` method. For example: `ip ntable change name arp_cache queue 20 dev eth0`.

You can also add entries to the ARP table by `arp add`. And it is possible to add static entries manually to the ARP table, like this: `arp -s <IPAddress> <MacAddress>`. The static ARP entries are not deleted by the neigbouring subsystem garbage collector, but they are not persistent over reboot.

The next section briefly describes how network events are handled in the neighbouring subsystem.

Handling Network Events

The neighbouring core does not register any events with the `register_netdevice_notifier()` method. On the other hand, the ARP module and the NDISC module do register network events. In ARP, the `arp_netdev_event()` method is registered as the callback for `netdev` events. It handles changes of MAC address events by calling the generic `neigh_changeaddr()` method and by calling the `rt_cache_flush()` method. From kernel 3.11, you handle a NETDEV_CHANGE event when there was a change of the IFF_NOARP flag by calling the `neigh_changeaddr()` method. A NETDEV_CHANGE event is triggered when a device changes its flags, by the `__dev_notify_flags()` method, or when a device changes its state, by the `netdev_state_change()` method. In NDISC, the `ndisc_netdev_event()` method is registered as the callback for netdev events; it handles the NETDEV_CHANGEADDR, NETDEV_DOWN, and NETDEV_NOTIFY_PEERS events.

After describing the fundamental data structures common to IPv4 and IPv6, like the neighbouring table (`neigh_table`) and the `neighbour` structure, and after discussing how a `neighbour` object is created and freed, it is time to describe the implementation of the first neighbouring protocol, the ARP protocol.

The ARP protocol (IPv4)

The ARP protocol is defined in RFC 826. When working with Ethernet, the addresses are called MAC addresses and are 48-bit values. MAC addresses should be unique, but you must take into account that you may encounter a non-unique MAC address. A common reason for this is that on most network interfaces, a system administrator can configure MAC addresses with userspace tools like `ifconfig` or `ip`.

When sending an IPv4 packet, you know the destination IPv4 address. You should build an Ethernet header, which should include a destination MAC address. Finding the MAC address based on a given IPv4 address is done by the ARP protocol as you will see shortly. If the MAC address is unknown, you send an ARP request as a broadcast. This ARP request contains the IPv4 address you are seeking. If there is a host with such an IPv4 address, this host sends a unicast ARP response as a reply. The ARP table (`arp_tbl`) is an instance of the `neigh_table` structure. The ARP header is represented by the `arphdr` structure:

```
struct arphdr {
    __be16          ar_hrd;         /* format of hardware address   */
    __be16          ar_pro;         /* format of protocol address   */
    unsigned char   ar_hln;         /* length of hardware address   */
    unsigned char   ar_pln;         /* length of protocol address   */
    __be16          ar_op;          /* ARP opcode (command)         */
#if 0
    *
    *      Ethernet looks like this : This bit is variable sized however...
    */
    unsigned char           ar_sha[ETH_ALEN];       /* sender hardware address      */
    unsigned char           ar_sip[4];              /* sender IP address            */
    unsigned char           ar_tha[ETH_ALEN];       /* target hardware address      */
    unsigned char           ar_tip[4];              /* target IP address            */
#endif
};
```

(include/uapi/linux/if_arp.h)

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

- ar_hrd is the hardware type; for Ethernet it is 0x01. For the full list of available ARP header hardware identifiers, see ARPHRD_XXX definitions in include/uapi/linux/if_arp.h.
- ar_pro is the protocol ID; for IPv4 it is 0x80. For the full list of available protocols IDs, see ETH_P_XXX in include/uapi/linux/if_ether.h.
- ar_hln is the hardware address length in bytes, which is 6 bytes for Ethernet addresses.
- ar_pln is the length of the protocol address in bytes, which is 4 bytes for IPv4 addresses.
- ar_op is the opcode, ARPOP_REQUEST for an ARP request, and ARPOP_REPLY for an ARP reply. For the full list of available ARP header opcodes look in include/uapi/linux/if_arp.h.

Immediately after the ar_op are the sender hardware (MAC) address and IPv4 address, and the target hardware (MAC) address and IPv4 address. These addresses are not part of the ARP header (arphdr) structure. In the arp_process() method, they are extracted by reading the corresponding offsets of the ARP header, as you can see in the explanation about the arp_process() method in the section "ARP: Receiving Solicitation Requests and Replies" later in this chapter. Figure 7-1 shows an ARP header for an ARP Ethernet packet.

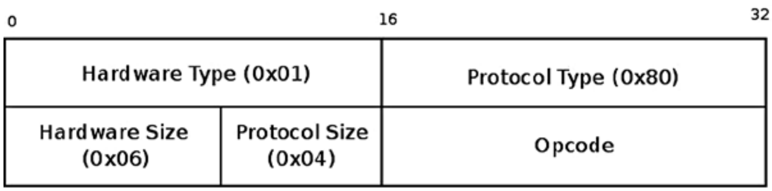

Figure 7-1. *ARP header (for Ethernet)*

In ARP, four neigh_ops objects are defined: arp_direct_ops, arp_generic_ops, arp_hh_ops, and arp_broken_ops. The initialization of the ARP table neigh_ops object is done by the arp_constructor() method, based on the network device features:

- If the header_ops of the net_device object is NULL, the neigh_ops object will be set to be arp_direct_ops. In this case, sending the packet will be done with the neigh_direct_output() method, which is in fact a wrapper around dev_queue_xmit(). In most Ethernet network devices, however, the header_ops of the net_device object is initialized to be eth_header_ops by the generic ether_setup() method; see net/ethernet/eth.c.
- If the header_ops of the net_device object contains a NULL cache() callback, then the neigh_ops object will be set to be arp_generic_ops.
- If the header_ops of the net_device object contains a non-NULL cache() callback, then the neigh_ops object will be set to be arp_hh_ops. In the case of using the generic eth_header_ops object, the cache() callback is the eth_header_cache() callback.
- For three types of devices, the neigh_ops object will be set to be arp_broken_ops (when the type of the net_device object is ARPHRD_ROSE, ARPHRD_AX25, or ARPHRD_NETROM).

Now that I've covered the ARP protocol and the ARP header (arphdr) object, let's look at how ARP solicitation requests are sent.

ARP: Sending Solicitation Requests

Where are solicitation requests being sent? The most common case is in the Tx path, before actually leaving the network layer (L3) and moving to the link layer (L2). In the `ip_finish_output2()` method, you first perform a lookup for the next hop IPv4 address in the ARP table by calling the `__ipv4_neigh_lookup_noref()` method, and if you don't find any matching neighbour entry, you create one by calling the `__neigh_create()` method:

```
static inline int ip_finish_output2(struct sk_buff *skb)
{
        struct dst_entry *dst = skb_dst(skb);
        struct rtable *rt = (struct rtable *)dst;
        struct net_device *dev = dst->dev;
        unsigned int hh_len = LL_RESERVED_SPACE(dev);
        struct neighbour *neigh;
        u32 nexthop;
        . . .
        . . .
        nexthop = (__force u32) rt_nexthop(rt, ip_hdr(skb)->daddr);
        neigh = __ipv4_neigh_lookup_noref(dev, nexthop);
        if (unlikely(!neigh))
                neigh = __neigh_create(&arp_tbl, &nexthop, dev, false);
        if (!IS_ERR(neigh)) {
                int res = dst_neigh_output(dst, neigh, skb);
     . . .
}
```

Let's take a look in the `dst_neigh_output()` method:

```
static inline int dst_neigh_output(struct dst_entry *dst, struct neighbour *n,
                                   struct sk_buff *skb)
{
        const struct hh_cache *hh;

        if (dst->pending_confirm) {
                unsigned long now = jiffies;

                dst->pending_confirm = 0;
                /* avoid dirtying neighbour */
                if (n->confirmed != now)
                        n->confirmed = now;
        }
```

When you reach this method for the first time with this flow, `nud_state` is not NUD_CONNECTED, and the output callback is the `neigh_resolve_output()` method:

```
        hh = &n->hh;
        if ((n->nud_state & NUD_CONNECTED) && hh->hh_len)
                return neigh_hh_output(hh, skb);
        else
                return n->output(n, skb);
}
```

(include/net/dst.h)

In the neigh_resolve_output() method, you call the neigh_event_send() method, which eventually puts the SKB in the arp_queue of the neighbour by __skb_queue_tail(&neigh->arp_queue, skb); later, the neigh_probe() method, invoked from the neighbour timer handler, neigh_timer_handler(), will send the packet by invoking the solicit() method (neigh->ops->solicit is the arp_solicit() method in our case):

```
static void neigh_probe(struct neighbour *neigh)
        __releases(neigh->lock)
{
        struct sk_buff *skb = skb_peek(&neigh->arp_queue);
        . . .
        neigh->ops->solicit(neigh, skb);
        atomic_inc(&neigh->probes);
        kfree_skb(skb);
}
```

Let's take a look at the arp_solicit() method, which actually sends the ARP request:

```
static void arp_solicit(struct neighbour *neigh, struct sk_buff *skb)
{
        __be32 saddr = 0;
        u8 dst_ha[MAX_ADDR_LEN], *dst_hw = NULL;
        struct net_device *dev = neigh->dev;
        __be32 target = *(__be32 *)neigh->primary_key;
        int probes = atomic_read(&neigh->probes);
        struct in_device *in_dev;

        rcu_read_lock();
        in_dev = __in_dev_get_rcu(dev);
        if (!in_dev) {
                rcu_read_unlock();
                return;
        }
```

With the arp_announce procfs entry, you can set restrictions for which local source IP address to use for the ARP packet you want to send:

- *0:* Use any local address, configured on any interface. This is the default value.
- *1:* First try to use addresses that are on the target subnet. If there are no such addresses, use level 2.
- *2:* Use primary IP address.

Note that the max value of these two entries is used:

```
/proc/sys/net/ipv4/conf/all/arp_announce
/proc/sys/net/ipv4/conf/<netdeviceName>/arp_announce
```

See also the description of the IN_DEV_ARP_ANNOUNCE macro in the “Quick Reference” section at the end of this chapter.

```
        switch (IN_DEV_ARP_ANNOUNCE(in_dev)) {
        default:
        case 0:         /* By default announce any local IP */
                if (skb && inet_addr_type(dev_net(dev),
                                          ip_hdr(skb)->saddr) == RTN_LOCAL)
                        saddr = ip_hdr(skb)->saddr;
                break;
        case 1:         /* Restrict announcements of saddr in same subnet */
                if (!skb)
                break;
                saddr = ip_hdr(skb)->saddr;
                if (inet_addr_type(dev_net(dev), saddr) == RTN_LOCAL) {
```

The inet_addr_onlink() method checks whether the specified target address and the specified source address are on the same subnet:

```
                /* saddr should be known to target */
                if (inet_addr_onlink(in_dev, target, saddr))
                        break;
        }
        saddr = 0;
        break;
case 2:         /* Avoid secondary IPs, get a primary/preferred one */
        break;
}
rcu_read_unlock();

if (!saddr)
```

The inet_select_addr() method returns the address of the first primary interface of the specified device whose scope is smaller than the specified scope (RT_SCOPE_LINK in this case), and which is in the same subnet as the target:

```
        saddr = inet_select_addr(dev, target, RT_SCOPE_LINK);

        probes -= neigh->parms->ucast_probes;
        if (probes < 0) {
                if (!(neigh->nud_state & NUD_VALID))
                        pr_debug("trying to ucast probe in NUD_INVALID\n");
                neigh_ha_snapshot(dst_ha, neigh, dev);
                dst_hw = dst_ha;
        } else {
                probes -= neigh->parms->app_probes;
                if (probes < 0) {
```

CONFIG_ARPD is set when working with the userspace ARP daemon; there are projects like OpenNHRP, which are based on ARPD. Next Hop Resolution Protocol (NHRP) is used to improve the efficiency of routing computer network traffic over Non-Broadcast, Multiple Access (NBMA) networks (I don't discuss the ARPD userspace daemon in this book):

```
#ifdef CONFIG_ARPD
                        neigh_app_ns(neigh);
#endif
                        return;
                }
        }
```

Now you call the `arp_send()` method to send an ARP request. Note that the last parameter, `target_hw`, is NULL. You do not yet know the target hardware (MAC) address. When calling `arp_send()` with `target_hw` as NULL, a broadcast ARP request is sent:

```
        arp_send(ARPOP_REQUEST, ETH_P_ARP, target, dev, saddr,
                 dst_hw, dev->dev_addr, NULL);
}
```

Let's take a look at the `arp_send()` method, which is quite short:

```
void arp_send(int type, int ptype, __be32 dest_ip,
              struct net_device *dev, __be32 src_ip,
              const unsigned char *dest_hw, const unsigned char *src_hw,
              const unsigned char *target_hw)
{
        struct sk_buff *skb;

        /*
         *      No arp on this interface.
         */
```

You must check whether the IFF_NOARP is supported on this network device. There are cases in which ARP is disabled: an administrator can disable ARP, for example, by `ifconfig eth1 -arp` or by `ip link set eth1 arp off`. Some network devices set the IFF_NOARP flag upon creation—for example, IPv4 tunnel devices, or PPP devices, which do not need ARP. See the `ipip_tunnel_setup()` method in `net/ipv4/ipip.c` or the `ppp_setup()` method in `drivers/net/ppp_generic.c`.

```
        if (dev->flags&IFF_NOARP)
                return;
```

The `arp_create()` method creates an SKB with an ARP header and initializes it according to the specified parameters:

```
        skb = arp_create(type, ptype, dest_ip, dev, src_ip,
                         dest_hw, src_hw, target_hw);
        if (skb == NULL)
                return;
```

The only thing the `arp_xmit()` method does is call `dev_queue_xmit()` by the NF_HOOK() macro:

```
        arp_xmit(skb);
}
```

Now it is time to learn how these ARP requests are processed and how ARP replies are processed.

ARP: Receiving Solicitation Requests and Replies

In IPv4, the `arp_rcv()` method is responsible for handling ARP packets, as mentioned earlier. Let's take a look at the `arp_rcv()` method:

```
static int arp_rcv(struct sk_buff *skb, struct net_device *dev,
                   struct packet_type *pt, struct net_device *orig_dev)
{
        const struct arphdr *arp;
```

If the network device on which the ARP packet was received has the IFF_NOARP flag set, or if the packet is not destined for the local machine, or if it is for a loopback device, then the packet should be dropped. You continue and make some more sanity checks, and if everything is okay, you proceed to the `arp_process()` method, which performs the real work of processing an ARP packet:

```
        if (dev->flags & IFF_NOARP ||
            skb->pkt_type == PACKET_OTHERHOST ||
            skb->pkt_type == PACKET_LOOPBACK)
                goto freeskb;
```

If the SKB is shared, you must clone it because it might be changed by someone else while being processed by the `arp_rcv()` method. The `skb_share_check()` method creates a clone of the SKB if it is shared (see Appendix A).

```
        skb = skb_share_check(skb, GFP_ATOMIC);
        if (!skb)
                goto out_of_mem;

        /* ARP header, plus 2 device addresses, plus 2 IP addresses.  */
        if (!pskb_may_pull(skb, arp_hdr_len(dev)))
                goto freeskb;

        arp = arp_hdr(skb);
```

The `ar_hln` of the ARP header represents the length of a hardware address, which should be 6 bytes for Ethernet header, and should be equal to the `addr_len` of the `net_device` object. The `ar_pln` of the ARP header represents the length of the protocol address and should be equal to the length of an IPv4 address, which is 4 bytes:

```
        if (arp->ar_hln != dev->addr_len || arp->ar_pln != 4)
                goto freeskb;

        memset(NEIGH_CB(skb), 0, sizeof(struct neighbour_cb));
        return NF_HOOK(NFPROTO_ARP, NF_ARP_IN, skb, dev, NULL, arp_process);
```

```
freeskb:
        kfree_skb(skb);
out_of_mem:
        return 0;
}
```

Handling ARP requests is not restricted to packets that have the local host as their destination. When the local host is configured as a proxy ARP, or as a private VLAN proxy ARP (see RFC 3069), you also handle packets which have a destination that is not the local host. Support for private VLAN proxy ARP was added in kernel 2.6.34.

In the `arp_process()` method, you handle only ARP requests or ARP responses. For ARP requests you perform a lookup in the routing subsystem by the `ip_route_input_noref()` method. If the ARP packet is for the local host (the `rt_type` of the routing entry is RTN_LOCAL), you proceed to check some conditions (described shortly). If all these checks pass, an ARP reply is sent back with the `arp_send()` method. If the ARP packet is not for the local host but should be forwarded (the `rt_type` of the routing entry is RTN_UNICAST), then you check some conditions (also described shortly), and if they are fulfilled you perform a lookup in the proxy ARP table by calling the `pneigh_lookup()` method.

You will now see the implementation details of the main ARP method which handles ARP requests, the `arp_process()` method.

The arp_process() Method

Let's take a look at the `arp_process()` method, where the real work is done:

```
static int arp_process(struct sk_buff *skb)
{
        struct net_device *dev = skb->dev;
        struct in_device *in_dev = __in_dev_get_rcu(dev);
        struct arphdr *arp;
        unsigned char *arp_ptr;
        struct rtable *rt;
        unsigned char *sha;
        __be32 sip, tip;
        u16 dev_type = dev->type;
        int addr_type;
        struct neighbour *n;
        struct net *net = dev_net(dev);

        /* arp_rcv below verifies the ARP header and verifies the device
         * is ARP'able.
         */

        if (in_dev == NULL)
                goto out;
```

Fetch the ARP header from the SKB (it is the network header, see the `arp_hdr()` method):

```
        arp = arp_hdr(skb);

        switch (dev_type) {
        default:
                if (arp->ar_pro != htons(ETH_P_IP) ||
```

```
                        htons(dev_type) != arp->ar_hrd)
                            goto out;
                break;
        case ARPHRD_ETHER:
                . . .
                if ((arp->ar_hrd != htons(ARPHRD_ETHER) &&
                     arp->ar_hrd != htons(ARPHRD_IEEE802)) ||
                    arp->ar_pro != htons(ETH_P_IP))
                        goto out;
                break;
                . . .
```

You want to handle only ARP requests or ARP responses in the `arp_process()` method, and discard all other packets:

```
        /* Understand only these message types */

        if (arp->ar_op != htons(ARPOP_REPLY) &&
            arp->ar_op != htons(ARPOP_REQUEST))
                goto out;

/*
 *      Extract fields
 */
        arp_ptr = (unsigned char *)(arp + 1);
```

The arp_process() Method—Extracting Headers:

Immediately after the ARP header, there are the following fields (see the ARP header definition above):

- `sha`: The source hardware address (the MAC address, which is 6 bytes).
- `sip`: The source IPv4 address (4 bytes).
- `tha`: The target hardware address (the MAC address, which is 6 bytes).
- `tip`: The target IPv4 address (4 bytes).

Extract the `sip` and `tip` addresses:

```
        sha     = arp_ptr;
        arp_ptr += dev->addr_len;
```

Set `sip` to be the source IPv4 address after advancing `arp_ptr` with the corresponding offset:

```
        memcpy(&sip, arp_ptr, 4);
        arp_ptr += 4;
        switch (dev_type) {
        . . .
        default:
                arp_ptr += dev->addr_len;
        }
```

Set `tip` to be the target IPv4 address after advancing `arp_ptr` with the corresponding offset:

```
memcpy(&tip, arp_ptr, 4);
```

Discard these two types of packets:

- Multicast packets
- Packets for the loopback device if the use of local routing with loopback addresses is disabled; see also the description of the IN_DEV_ROUTE_LOCALNET macro in the "Quick Reference" section at the end of this chapter.

```
/*
 *      Check for bad requests for 127.x.x.x and requests for multicast
 *      addresses.  If this is one such, delete it.
 */
        if (ipv4_is_multicast(tip) ||
            (!IN_DEV_ROUTE_LOCALNET(in_dev) && ipv4_is_loopback(tip)))
                goto out;

        . . .
```

The source IP (`sip`) is 0 when you use Duplicate Address Detection (DAD). DAD lets you detect the existence of double L3 addresses on different hosts on a LAN. DAD is implemented in IPv6 as an integral part of the address configuration process, but not in IPv4. However, there is support for correctly handling DAD requests in IPv4, as you will soon see. The `arping` utility of the `iputils` package is an example for using DAD in IPv4. When sending ARP request with `arping -D`, you send an ARP request where the `sip` of the ARP header is 0. (The `-D` modifier tells `arping` to be in DAD mode); the `tip` is usually the sender IPv4 address (because you want to check whether there is another host on the same LAN with the same IPv4 address as yours); if there is a host with the same IP address as the `tip` of the DAD ARP request, it will send back an ARP reply (without adding the sender to its neighbouring table):

```
        /* Special case: IPv4 duplicate address detection packet (RFC2131) */
        if (sip == 0) {
                if (arp->ar_op == htons(ARPOP_REQUEST) &&
```

The arp_process() Method—arp_ignore() and arp_filter() Methods

The `arp_ignore procfs` entry provides support for different modes for sending ARP replies as a response for an ARP request. The value used is the max value of `/proc/sys/net/ipv4/conf/all/arp_ignore` and `/proc/sys/net/ipv4/conf/<netDeviceName>/arp_ignore`. By default, the value of the `arp_ignore procfs` entry is 0, and in such a case, the `arp_ignore()` method returns 0. You reply to the ARP request with `arp_send()`, as you can see in the next code snippet (assuming that `inet_addr_type(net, tip)` returned RTN_LOCAL). The `arp_ignore()` method checks the value of IN_DEV_ARP_IGNORE(in_dev); for more details, see the `arp_ignore()` implementation in `net/ipv4/arp.c` and the description of the IN_DEV_ARP_IGNORE macro in the "Quick Reference" section at the end of this chapter:

```
                 inet_addr_type(net, tip) == RTN_LOCAL &&
                !arp_ignore(in_dev, sip, tip))
                arp_send(ARPOP_REPLY, ETH_P_ARP, sip, dev, tip, sha,
                         dev->dev_addr, sha);
        goto out;
}
```

```
if (arp->ar_op == htons(ARPOP_REQUEST) &&
    ip_route_input_noref(skb, tip, sip, 0, dev) == 0) {

        rt = skb_rtable(skb);
        addr_type = rt->rt_type;
```

When addr_type equals RTN_LOCAL, the packet is for local delivery:

```
        if (addr_type == RTN_LOCAL) {
                int dont_send;

                dont_send = arp_ignore(in_dev, sip, tip);
```

The arp_filter() method fails (returns 1) in two cases:

- When the lookup in the routing tables with the ip_route_output() method fails.
- When the outgoing network device of the routing entry is different than the network device on which the ARP request was received.

In case of success, the arp_filter() method returns 0 (see also the description of the IN_DEV_ARPFILTER macro in the “Quick Reference” section at the end of this chapter):

```
                     if (!dont_send && IN_DEV_ARPFILTER(in_dev))
                        dont_send = arp_filter(sip, tip, dev);
                if (!dont_send) {
```

Before sending the ARP reply, you want to add the sender to your neighbouring table or update it; this is done with the neigh_event_ns() method. The neigh_event_ns() method creates a new neighbouring table entry and sets its state to be NUD_STALE. If there is already such an entry, it updates its state to be NUD_STALE, with the neigh_update() method. Adding entries this way is termed *passive learning:*

```
                        n = neigh_event_ns(&arp_tbl, sha, &sip, dev);
                        if (n) {
                                arp_send(ARPOP_REPLY, ETH_P_ARP, sip,
                                         dev, tip, sha, dev->dev_addr,
                                         sha);
                                neigh_release(n);
                        }
                }
                goto out;
        } else if (IN_DEV_FORWARD(in_dev)) {
```

The arp_fwd_proxy() method returns 1 when the device can be used as an ARP proxy; the arp_fwd_pvlan() method returns 1 when the device can be used as an ARP VLAN proxy:

```
               if (addr_type == RTN_UNICAST  &&
                    (arp_fwd_proxy(in_dev, dev, rt) ||
                     arp_fwd_pvlan(in_dev, dev, rt, sip, tip) ||
                     (rt->dst.dev != dev &&
                      pneigh_lookup(&arp_tbl, net, &tip, dev, 0)))) {
```

Again, call the `neigh_event_ns()` method to create a neighbour entry of the sender with NUD_STALE, or if such an entry exists, update that entry state to be NUD_STALE:

```
                                n = neigh_event_ns(&arp_tbl, sha, &sip, dev);
                                if (n)
                                        neigh_release(n);

                                if (NEIGH_CB(skb)->flags & LOCALLY_ENQUEUED ||
                                    skb->pkt_type == PACKET_HOST ||
                                    in_dev->arp_parms->proxy_delay == 0) {
                                        arp_send(ARPOP_REPLY, ETH_P_ARP, sip,
                                                 dev, tip, sha, dev->dev_addr,
                                                 sha);
                                } else {
```

Delay sending an ARP reply by putting the SKB at the tail of the `proxy_queue`, by calling the `pneigh_enqueue()` method. Note that the delay is random and is a number between 0 and `in_dev->arp_parms->proxy_delay`:

```
                                        pneigh_enqueue(&arp_tbl,
                                                       in_dev->arp_parms, skb);
                                        return 0;
                                }
                                goto out;
                        }
                }
        }

        /* Update our ARP tables */
```

Note that the last parameter of calling the `__neigh_lookup()` method is 0, which means that you only perform a lookup in the neighbouring table (and do not create a new neighbour if the lookup failed):

```
        n = __neigh_lookup(&arp_tbl, &sip, dev, 0);
```

The IN_DEV_ARP_ACCEPT macro tells you whether the network device is set to accept ARP requests (see also the description of the IN_DEV_ARP_ACCEPT macro in the “Quick Reference” section at the end of this of this chapter):

```
        if (IN_DEV_ARP_ACCEPT(in_dev)) {
                /* Unsolicited ARP is not accepted by default.
                   It is possible, that this option should be enabled for some
                   devices (strip is candidate)
                */
```

Unsolicited ARP requests are sent only to update the neighbouring table. In such requests, `tip` is equal to `sip` (the `arping` utility supports sending unsolicited ARP requests by `arping -U`):

```
        if (n == NULL &&
            (arp->ar_op == htons(ARPOP_REPLY) ||
             (arp->ar_op == htons(ARPOP_REQUEST) && tip == sip)) &&
           inet_addr_type(net, sip) == RTN_UNICAST)
               n = __neigh_lookup(&arp_tbl, &sip, dev, 1);
}
```

```
if (n) {
        int state = NUD_REACHABLE;
        int override;

        /* If several different ARP replies follows back-to-back,
           use the FIRST one. It is possible, if several proxy
           agents are active. Taking the first reply prevents
           arp trashing and chooses the fastest router.
        */
        override = time_after(jiffies, n->updated + n->parms->locktime);

        /* Broadcast replies and request packets
           do not assert neighbour reachability.
         */
         if (arp->ar_op != htons(ARPOP_REPLY) ||
             skb->pkt_type != PACKET_HOST)
                 state = NUD_STALE;
```

Call `neigh_update()` to update the neighbouring table:

```
                neigh_update(n, sha, state,
                             override ? NEIGH_UPDATE_F_OVERRIDE : 0);
                neigh_release(n);
        }

out:
        consume_skb(skb);
        return 0;
}
```

Now that you know about the IPv4 ARP protocol implementation, it is time to move on to IPv6 NDISC protocol implementation. You will soon notice some of the differences between the neighbouring subsystem implementation in IPv4 and in IPv6.

The NDISC Protocol (IPv6)

The Neighbour Discovery (NDISC) protocol is based on RFC 2461, "Neighbour Discovery for IP Version 6 (IPv6)," which was later obsoleted by RFC 4861 from 2007. IPv6 nodes (hosts or routers) on the same link use the Neighbour Discovery protocol to discover each other's presence, to discover routers, to determine each other's L2 addresses, and to maintain neighbour reachability information. Duplicate Address Detection (DAD) was added to avoid double L3 addresses on the same LAN. I discuss DAD and handling NDISC neighbour solicitation and neighbour advertisements shortly.

Next you learn how IPv6 neighbour discovery protocols avoid creating duplicate IPv6 addresses.

Duplicate Address Detection (DAD)

How can you be sure there is no other same IPv6 address on a LAN? The chances are low, but if such address does exist, it may cause trouble. DAD is a solution. When a host tries to configure an address, it first creates a Link Local address (a Link Local address starts with FE80). This address is tentative (IFA_F_TENTATIVE), which means that the host can communicate only with ND messages. Then the host starts the DAD process by calling the `addrconf_dad_start()` method (`net/ipv6/addrconf.c`). The host sends a Neighbour Solicitation DAD message. The target is its tentative address, the source is all zeros (the unspecified address). If there is no answer in a specified time interval, the state is changed to permanent (IFA_F_PERMANENT). When Optimistic DAD (CONFIG_IPV6_OPTIMISTIC_DAD) is set, you don't wait until DAD is completed, but allow hosts to communicate with peers before DAD has finished successfully. See RFC 4429, "Optimistic Duplicate Address Detection (DAD) for IPv6," from 2006.

The neighbouring table for IPv6 is called `nd_tbl`:

```
struct neigh_table nd_tbl = {
        .family =       AF_INET6,
        .key_len =      sizeof(struct in6_addr),
        .hash =         ndisc_hash,
        .constructor =  ndisc_constructor,
        .pconstructor = pndisc_constructor,
        .pdestructor =  pndisc_destructor,
        .proxy_redo =   pndisc_redo,
        .id =           "ndisc_cache",
        .parms = {
                .tbl                    = &nd_tbl,
                .base_reachable_time    = ND_REACHABLE_TIME,
                .retrans_time           = ND_RETRANS_TIMER,
                .gc_staletime           = 60 * HZ,
                .reachable_time         = ND_REACHABLE_TIME,
                .delay_probe_time       = 5 * HZ,
                .queue_len_bytes        = 64*1024,
                .ucast_probes           = 3,
                .mcast_probes           = 3,
                .anycast_delay          = 1 * HZ,
                .proxy_delay            = (8 * HZ) / 10,
                .proxy_qlen             = 64,
        },
        .gc_interval =    30 * HZ,
        .gc_thresh1 =    128,
        .gc_thresh2 =    512,
        .gc_thresh3 =   1024,
};
(net/ipv6/ndisc.c)
```

Note that some of the members of the NDISC table are equal to the parallel members in the ARP table—for example, the values of the garbage collector thresholds (`gc_thresh1`, `gc_thresh2` and `gc_thresh3`).

The Linux IPv6 Neighbour Discovery implementation is based on ICMPv6 messages to manage the interaction between neighbouring nodes. The Neighbour Discovery protocol defines the following five ICMPv6 message types:

```
#define NDISC_ROUTER_SOLICITATION       133
#define NDISC_ROUTER_ADVERTISEMENT      134
#define NDISC_NEIGHBOUR_SOLICITATION    135
#define NDISC_NEIGHBOUR_ADVERTISEMENT   136
#define NDISC_REDIRECT                  137

(include/net/ndisc.h)
```

Note that these five ICMPv6 message types are informational messages. ICMPv6 message types whose values are in the range from 0 to 127 are error messages, and ICMPv6 message types whose values are from 128 to 255 are informational messages. For more on that, see Chapter 3, which discusses the ICMP protocol. This chapter discusses only the Neighbour Solicitation and the Neighbour Discovery messages.

As mentioned in the beginning of this chapter, because neighbouring discovery messages are ICMPv6 messages, they are handled by the `icmpv6_rcv()` method, which in turn invokes the `ndisc_rcv()` method for ICMPv6 packets whose message type is one of the five types mentioned earlier (see `net/ipv6/icmp.c`).

In NDISC, there are three `neigh_ops` objects: `ndisc_generic_ops`, `ndisc_hh_ops`, and `ndisc_direct_ops`:

- If the `header_ops` of the `net_device` object is NULL, the `neigh_ops` object will be set to be `ndisc_direct_ops`. As in the case of `arp_direct_ops`, sending the packet is done with the `neigh_direct_output()` method, which is in fact a wrapper around `dev_queue_xmit()`. Note that, as mentioned in the ARP section earlier, in most Ethernet network devices, the `header_ops` of the `net_device` object is not NULL.
- If the `header_ops` of the `net_device` object contains a NULL `cache()` callback, then the `neigh_ops` object is set to be `ndisc_generic_ops`.
- If the `header_ops` of the `net_device` object contains a non-NULL `cache()` callback, then the `neigh_ops` object is set to be `ndisc_hh_ops`.

This section discussed the DAD mechanism and how it helps to avoid duplicate addresses. The next section describes how solicitation requests are sent.

NIDSC: Sending Solicitation Requests

Similarly to what you saw in IPv6, you also perform a lookup and create an entry if you did not find any match:

```
static int ip6_finish_output2(struct sk_buff *skb)
{
        struct dst_entry *dst = skb_dst(skb);
        struct net_device *dev = dst->dev;
        struct neighbour *neigh;
        struct in6_addr *nexthop;
        int ret;
                . . .

                . . .

        nexthop = rt6_nexthop((struct rt6_info *)dst, &ipv6_hdr(skb)->daddr);
        neigh = __ipv6_neigh_lookup_noref(dst->dev, nexthop);
        if (unlikely(!neigh))
                neigh = __neigh_create(&nd_tbl, nexthop, dst->dev, false);
        if (!IS_ERR(neigh)) {
                ret = dst_neigh_output(dst, neigh, skb);
                . . .
```

Eventually, much like in the IPv4 Tx path, you call the solicit method `neigh->ops->solicit(neigh, skb)` from the `neigh_probe()` method. The `neigh->ops->solicit` in this case is the `ndisc_solicit()` method. The `ndisc_solicit()` is a very short method; it is in fact a wrapper around the `ndisc_send_ns()` method:

```
static void ndisc_solicit(struct neighbour *neigh, struct sk_buff *skb)
{
        struct in6_addr *saddr = NULL;
        struct in6_addr mcaddr;
```

```
        struct net_device *dev = neigh->dev;
        struct in6_addr *target = (struct in6_addr *)&neigh->primary_key;
        int probes = atomic_read(&neigh->probes);

        if (skb && ipv6_chk_addr(dev_net(dev), &ipv6_hdr(skb)->saddr, dev, 1))
                saddr = &ipv6_hdr(skb)->saddr;

        if ((probes -= neigh->parms->ucast_probes) < 0) {
                if (!(neigh->nud_state & NUD_VALID)) {
                        ND_PRINTK(1, dbg,
                                  "%s: trying to ucast probe in NUD_INVALID: %pI6\n",
                                  __func__, target);
                }
                ndisc_send_ns(dev, neigh, target, target, saddr);
        } else if ((probes -= neigh->parms->app_probes) < 0) {
#ifdef CONFIG_ARPD
                neigh_app_ns(neigh);
#endif
        } else {
                addrconf_addr_solict_mult(target, &mcaddr);
                ndisc_send_ns(dev, NULL, target, &mcaddr, saddr);
        }
}
```

In order to send the solicitation request, we need to build an `nd_msg` object:

```
struct nd_msg {
        struct icmp6hdr icmph;
        struct in6_addr target;
        __u8            opt[0];
};
```

(include/net/ndisc.h)

For a solicitation request, the ICMPv6 header type should be set to NDISC_NEIGHBOUR_SOLICITATION, and for solicitation reply, the ICMPv6 header type should be set to NDISC_NEIGHBOUR_ADVERTISEMENT. Note that with Neighbour Advertisement messages, there are cases when you need to set flags in the ICMPv6 header. The ICMPv6 header includes a structure named `icmpv6_nd_advt`, which includes the override, solicited, and router flags:

```
struct icmp6hdr {
        __u8            icmp6_type;
        __u8            icmp6_code;
        __sum16         icmp6_cksum;
        union {
                . . .
                . . .
                struct icmpv6_nd_advt {
#if defined(__LITTLE_ENDIAN_BITFIELD)
                        __u32           reserved:5,
                                        override:1,
                                        solicited:1,
```

```
                                                router:1,
                                                reserved2:24;
. . .
#endif
                } u_nd_advt;
        } icmp6_dataun;
. . .
#define icmp6_router            icmp6_dataun.u_nd_advt.router
#define icmp6_solicited         icmp6_dataun.u_nd_advt.solicited
#define icmp6_override          icmp6_dataun.u_nd_advt.override
. . .
```

(include/uapi/linux/icmpv6.h)

- When a message is sent in response to a Neighbour Solicitation, you set the solicited flag (icmp6_solicited).
- When you want to override a neighbouring cache entry (update the L2 address), you set the override flag (icmp6_override).
- When the host sending the Neighbour Advertisement message is a router, you set the router flag (icmp6_router).

You can see the use of these three flags in the ndisc_send_na() method that follows. Let's take a look at the ndisc_send_ns() method:

```
void ndisc_send_ns(struct net_device *dev, struct neighbour *neigh,
                   const struct in6_addr *solicit,
                   const struct in6_addr *daddr, const struct in6_addr *saddr)
{
        struct sk_buff *skb;
        struct in6_addr addr_buf;
        int inc_opt = dev->addr_len;
        int optlen = 0;
        struct nd_msg *msg;

        if (saddr == NULL) {
                if (ipv6_get_lladdr(dev, &addr_buf,
                                   (IFA_F_TENTATIVE|IFA_F_OPTIMISTIC)))
                        return;
                saddr = &addr_buf;
        }

        if (ipv6_addr_any(saddr))
                inc_opt = 0;
        if (inc_opt)
                optlen += ndisc_opt_addr_space(dev);

        skb = ndisc_alloc_skb(dev, sizeof(*msg) + optlen);
        if (!skb)
                return;
```

Build the ICMPv6 header, which is embedded in the nd_msg object:

```
        msg = (struct nd_msg *)skb_put(skb, sizeof(*msg));
        *msg = (struct nd_msg) {
                .icmph = {
                        .icmp6_type = NDISC_NEIGHBOUR_SOLICITATION,
                },
                .target = *solicit,
        };

        if (inc_opt)
                ndisc_fill_addr_option(skb, ND_OPT_SOURCE_LL_ADDR,
                                       dev->dev_addr);

        ndisc_send_skb(skb, daddr, saddr);
}
```

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

```
static void ndisc_send_na(struct net_device *dev, struct neighbour *neigh,
                          const struct in6_addr *daddr,
                          const struct in6_addr *solicited_addr,
                          bool router, bool solicited, bool override, bool inc_opt)
{
        struct sk_buff *skb;
        struct in6_addr tmpaddr;
        struct inet6_ifaddr *ifp;
        const struct in6_addr *src_addr;
        struct nd_msg *msg;
        int optlen = 0;

        . . .

        skb = ndisc_alloc_skb(dev, sizeof(*msg) + optlen);
        if (!skb)
                return;
```

Build the ICMPv6 header, which is embedded in the nd_msg object:

```
        msg = (struct nd_msg *)skb_put(skb, sizeof(*msg));
        *msg = (struct nd_msg) {
                .icmph = {
                        .icmp6_type = NDISC_NEIGHBOUR_ADVERTISEMENT,
                        .icmp6_router = router,
                        .icmp6_solicited = solicited,
                        .icmp6_override = override,
                },
                .target = *solicited_addr,
        };
```

```
        if (inc_opt)
                ndisc_fill_addr_option(skb, ND_OPT_TARGET_LL_ADDR,
                                       dev->dev_addr);

        ndisc_send_skb(skb, daddr, src_addr);
}
```

This section described how solicitation requests are sent. The next section talks about how Neighbour Solicitations and Advertisements are handled.

NDISC: Receiving Neighbour Solicitations and Advertisements

As mentioned, the `ndisc_rcv()` method handles all five neighbour discovery message types; let's take a look at this method:

```
int ndisc_rcv(struct sk_buff *skb)
{
        struct nd_msg *msg;

        if (skb_linearize(skb))
                return 0;

        msg = (struct nd_msg *)skb_transport_header(skb);

        __skb_push(skb, skb->data - skb_transport_header(skb));
```

According to RFC 4861, the hop limit of neighbour messages should be 255; the hop limit length is 8 bits, so the maximum hop limit is 255. A value of 255 assures that the packet was not forwarded, and this assures you that you are not exposed to some security attack. Packets that do not fulfill this requirement are discarded:

```
        if (ipv6_hdr(skb)->hop_limit != 255) {
                ND_PRINTK(2, warn, "NDISC: invalid hop-limit: %d\n",
                          ipv6_hdr(skb)->hop_limit);
                return 0;
        }
```

According to RFC 4861, the ICMPv6 code of neighbour messages should be 0, so drop packets that do not fulfill this requirement:

```
        if (msg->icmph.icmp6_code != 0) {
                ND_PRINTK(2, warn, "NDISC: invalid ICMPv6 code: %d\n",
                          msg->icmph.icmp6_code);
        return 0;
        }

        memset(NEIGH_CB(skb), 0, sizeof(struct neighbour_cb));

        switch (msg->icmph.icmp6_type) {
        case NDISC_NEIGHBOUR_SOLICITATION:
                ndisc_recv_ns(skb);
                break;
```

```
        case NDISC_NEIGHBOUR_ADVERTISEMENT:
                ndisc_recv_na(skb);
                break;

        case NDISC_ROUTER_SOLICITATION:
                ndisc_recv_rs(skb);
                break;

        case NDISC_ROUTER_ADVERTISEMENT:
                ndisc_router_discovery(skb);
                break;

        case NDISC_REDIRECT:
                ndisc_redirect_rcv(skb);
                break;
        }

        return 0;
}
```

I do not discuss router solicitations and router advertisements in this chapter, since they are discussed in Chapter 8. Let's take a look at the `ndisc_recv_ns()` method:

```
static void ndisc_recv_ns(struct sk_buff *skb)
{
        struct nd_msg *msg = (struct nd_msg *)skb_transport_header(skb);
        const struct in6_addr *saddr = &ipv6_hdr(skb)->saddr;
        const struct in6_addr *daddr = &ipv6_hdr(skb)->daddr;
        u8 *lladdr = NULL;
        u32 ndoptlen = skb->tail - (skb->transport_header +
                                    offsetof(struct nd_msg, opt));
        struct ndisc_options ndopts;
        struct net_device *dev = skb->dev;
        struct inet6_ifaddr *ifp;
        struct inet6_dev *idev = NULL;
        struct neighbour *neigh;
```

The `ipv6_addr_any()` method returns 1 when `saddr` is the unspecified address of all zeroes (IPV6_ADDR_ANY). When the source address is the unspecified address (all zeroes), this means that the request is DAD:

```
        int dad = ipv6_addr_any(saddr);
        bool inc;
        int is_router = -1;
```

Perform some validity checks:

```
        if (skb->len < sizeof(struct nd_msg)) {
                ND_PRINTK(2, warn, "NS: packet too short\n");
                return;
        }
```

```
if (ipv6_addr_is_multicast(&msg->target)) {
        ND_PRINTK(2, warn, "NS: multicast target address\n");
        return;
}

/*
 * RFC2461 7.1.1:
 * DAD has to be destined for solicited node multicast address.
 */
if (dad && !ipv6_addr_is_solict_mult(daddr)) {
        ND_PRINTK(2, warn, "NS: bad DAD packet (wrong destination)\n");
        return;
}

if (!ndisc_parse_options(msg->opt, ndoptlen, &ndopts)) {
        ND_PRINTK(2, warn, "NS: invalid ND options\n");
        return;
}

if (ndopts.nd_opts_src_lladdr) {
        lladdr = ndisc_opt_addr_data(ndopts.nd_opts_src_lladdr, dev);
        if (!lladdr) {
                ND_PRINTK(2, warn,
                          "NS: invalid link-layer address length\n");
                return;
        }

        /* RFC2461 7.1.1:
         *      If the IP source address is the unspecified address,
         *      there MUST NOT be source link-layer address option
         *      in the message.
         */
        if (dad) {
                ND_PRINTK(2, warn,
                          "NS: bad DAD packet (link-layer address option)\n");
                return;
        }
}

inc = ipv6_addr_is_multicast(daddr);

ifp = ipv6_get_ifaddr(dev_net(dev), &msg->target, dev, 1);
if (ifp) {

        if (ifp->flags & (IFA_F_TENTATIVE|IFA_F_OPTIMISTIC)) {
                if (dad) {
                        /*
                         * We are colliding with another node
                         * who is doing DAD
                         * so fail our DAD process
                         */
```

```
                        addrconf_dad_failure(ifp);
                        return;
                } else {
                        /*
                         * This is not a dad solicitation.
                         * If we are an optimistic node,
                         * we should respond.
                         * Otherwise, we should ignore it.
                        */
                        if (!(ifp->flags & IFA_F_OPTIMISTIC))
                                goto out;
                }
        }

        idev = ifp->idev;
} else {
        struct net *net = dev_net(dev);

        idev = in6_dev_get(dev);
        if (!idev) {
                /* XXX: count this drop? */
                return;
        }

        if (ipv6_chk_acast_addr(net, dev, &msg->target) ||
            (idev->cnf.forwarding &&
             (net->ipv6.devconf_all->proxy_ndp || idev->cnf.proxy_ndp) &&
             (is_router = pndisc_is_router(&msg->target, dev)) >= 0)) {
                if (!(NEIGH_CB(skb)->flags & LOCALLY_ENQUEUED) &&
                    skb->pkt_type != PACKET_HOST &&
                    inc != 0 &&
                    idev->nd_parms->proxy_delay != 0) {
                        /*
                         * for anycast or proxy,
                         * sender should delay its response
                         * by a random time between 0 and
                         * MAX_ANYCAST_DELAY_TIME seconds.
                         * (RFC2461) -- yoshfuji
                        */
                        struct sk_buff *n = skb_clone(skb, GFP_ATOMIC);
                        if (n)
                                pneigh_enqueue(&nd_tbl, idev->nd_parms, n);
                        goto out;
                }
        } else
                goto out;
}
```

```
if (is_router < 0)
        is_router = idev->cnf.forwarding;

if (dad) {
```

Send a neighbour advertisement message:

```
        ndisc_send_na(dev, NULL, &in6addr_linklocal_allnodes, &msg->target,
                      !!is_router, false, (ifp != NULL), true);
        goto out;
}

if (inc)
        NEIGH_CACHE_STAT_INC(&nd_tbl, rcv_probes_mcast);
else
        NEIGH_CACHE_STAT_INC(&nd_tbl, rcv_probes_ucast);

/*
 *      update / create cache entry
 *      for the source address
*/
neigh = __neigh_lookup(&nd_tbl, saddr, dev,
                       !inc || lladdr || !dev->addr_len);
if (neigh)
```

Update your neighbouring table with the sender's L2 address; the `nud_state` will be set to be NUD_STALE:

```
        neigh_update(neigh, lladdr, NUD_STALE,
                     NEIGH_UPDATE_F_WEAK_OVERRIDE|
                     NEIGH_UPDATE_F_OVERRIDE);
if (neigh || !dev->header_ops) {
```

Send a Neighbour Advertisement message:

```
                ndisc_send_na(dev, neigh, saddr, &msg->target,
                              !!is_router,
                              true, (ifp != NULL && inc), inc);
                if (neigh)
                        neigh_release(neigh);
        }

out:
        if (ifp)
                in6_ifa_put(ifp);
        else
                in6_dev_put(idev);
}
```

Let's take a look at the method that handles Neighbour Advertisements, ndisc_recv_na():

```
static void ndisc_recv_na(struct sk_buff *skb)
{
        struct nd_msg *msg = (struct nd_msg *)skb_transport_header(skb);
        const struct in6_addr *saddr = &ipv6_hdr(skb)->saddr;
        const struct in6_addr *daddr = &ipv6_hdr(skb)->daddr;
        u8 *lladdr = NULL;
        u32 ndoptlen = skb->tail - (skb->transport_header +
                                    offsetof(struct nd_msg, opt));
        struct ndisc_options ndopts;
        struct net_device *dev = skb->dev;
        struct inet6_ifaddr *ifp;
        struct neighbour *neigh;

        if (skb->len < sizeof(struct nd_msg)) {
                ND_PRINTK(2, warn, "NA: packet too short\n");
                return;
        }

        if (ipv6_addr_is_multicast(&msg->target)) {
                ND_PRINTK(2, warn, "NA: target address is multicast\n");
                return;
        }

        if (ipv6_addr_is_multicast(daddr) &&
            msg->icmph.icmp6_solicited) {
                ND_PRINTK(2, warn, "NA: solicited NA is multicasted\n");
                return;
        }

        if (!ndisc_parse_options(msg->opt, ndoptlen, &ndopts)) {
                ND_PRINTK(2, warn, "NS: invalid ND option\n");
                return;
        }
        if (ndopts.nd_opts_tgt_lladdr) {
                lladdr = ndisc_opt_addr_data(ndopts.nd_opts_tgt_lladdr, dev);
                if (!lladdr) {
                        ND_PRINTK(2, warn,
                                  "NA: invalid link-layer address length\n");
                        return;
                }
        }
        ifp = ipv6_get_ifaddr(dev_net(dev), &msg->target, dev, 1);
        if (ifp) {
                if (skb->pkt_type != PACKET_LOOPBACK
                    && (ifp->flags & IFA_F_TENTATIVE)) {
                                addrconf_dad_failure(ifp);
                                return;
                }
                /* What should we make now? The advertisement
                   is invalid, but ndisc specs say nothing
```

```
                   about it. It could be misconfiguration, or
                   an smart proxy agent tries to help us :-)

                   We should not print the error if NA has been
                   received from loopback - it is just our own
                   unsolicited advertisement.
                 */
                if (skb->pkt_type != PACKET_LOOPBACK)
                        ND_PRINTK(1, warn,
                                  "NA: someone advertises our address %pI6 on %s!\n",
                                  &ifp->addr, ifp->idev->dev->name);
                in6_ifa_put(ifp);
                return;
        }
        neigh = neigh_lookup(&nd_tbl, &msg->target, dev);

        if (neigh) {
                u8 old_flags = neigh->flags;
                struct net *net = dev_net(dev);

                if (neigh->nud_state & NUD_FAILED)
                        goto out;

                /*
                 * Don't update the neighbour cache entry on a proxy NA from
                 * ourselves because either the proxied node is off link or it
                 * has already sent a NA to us.
                 */
                if (lladdr && !memcmp(lladdr, dev->dev_addr, dev->addr_len) &&
                    net->ipv6.devconf_all->forwarding &&
                    net->ipv6.devconf_all->proxy_ndp &&
                    pneigh_lookup(&nd_tbl, net, &msg->target, dev, 0)) {
                        /* XXX: idev->cnf.proxy_ndp */
                        goto out;
                }
```

Update the neighbouring table. When the received message is a Neighbour Solicitation, the `icmp6_solicited` is set, so you want to set the state to be NUD_REACHABLE. When the `icmp6_override` flag is set, you want the `override` flag to be set (this mean update the L2 address with the specified `lladdr`, if it is different):

```
                neigh_update(neigh, lladdr,
                             msg->icmph.icmp6_solicited ? NUD_REACHABLE : NUD_STALE,
                             NEIGH_UPDATE_F_WEAK_OVERRIDE|
                             (msg->icmph.icmp6_override ? NEIGH_UPDATE_F_OVERRIDE : 0)|
                             NEIGH_UPDATE_F_OVERRIDE_ISROUTER|
                             (msg->icmph.icmp6_router ? NEIGH_UPDATE_F_ISROUTER : 0));

                if ((old_flags & ~neigh->flags) & NTF_ROUTER) {
                        /*
                         * Change: router to host
                         */
```

```
                        struct rt6_info *rt;
                        rt = rt6_get_dflt_router(saddr, dev);
                        if (rt)
                                ip6_del_rt(rt);
                }

out:
                neigh_release(neigh);
        }
}
```

Summary

This chapter described the neighbouring subsystem in IPv4 and in IPv6. First you learned about the goals of the neighbouring subsystem. Then you learned about ARP requests and ARP replies in IPv4, and about NDISC Neighbour Solicitation and NDISC Neighbour Advertisements in IPv6. You also found out about how DAD implementation avoids duplicate IPv6 addresses, and you saw various methods for handling the neighbouring subsystem requests and replies. Chapter 8 discusses the IPv6 subsystem implementation. The "Quick Reference" section that follows covers the top methods and macros related to the topics discussed in this chapter, ordered by their context. I also show the `neigh_statistics` structure, which represents statistics collected by the neighbouring subsystem.

Quick Reference

The following are some important methods and macros of the neighbouring subsystem, and a description of the `neigh_statistics` structure.

■ **Note** The core neighbouring code is in `net/core/neighbour.c`, `include/net/neighbour.h` and `include/uapi/linux/neighbour.h`.

The ARP code (IPv4) is in `net/ipv4/arp.c`, `include/net/arp.h` and in `include/uapi/linux/if_arp.h`.

The NDISC code (IPv6) is in `net/ipv6/ndisc.c` and `include/net/ndisc.h`.

Methods

Let's start by covering the methods.

void neigh_table_init(struct neigh_table *tbl)

This method invokes the `neigh_table_init_no_netlink()` method to perform the initialization of the neighbouring table, and links the table to the global neighbouring tables linked list (`neigh_tables`).

void neigh_table_init_no_netlink(struct neigh_table *tbl)

This method performs all the neighbour initialization apart from linking it to the global neighbouring table linked list, which is done by the `neigh_table_init()`, as mentioned earlier.

int neigh_table_clear(struct neigh_table *tbl)

This method frees the resources of the specified neighbouring table.

struct neighbour *neigh_alloc(struct neigh_table *tbl, struct net_device *dev)

This method allocates a neighbour object.

struct neigh_hash_table *neigh_hash_alloc(unsigned int shift)

This method allocates a neighbouring hash table.

struct neighbour *__neigh_create(struct neigh_table *tbl, const void *pkey, struct net_device *dev, bool want_ref)

This method creates a neighbour object.

int neigh_add(struct sk_buff *skb, struct nlmsghdr *nlh, void *arg)

This method adds a neighbour entry; it is the handler for netlink RTM_NEWNEIGH message.

int neigh_delete(struct sk_buff *skb, struct nlmsghdr *nlh, void *arg)

This method deletes a neighbour entry; it is the handler for netlink RTM_DELNEIGH message.

void neigh_probe(struct neighbour *neigh)

This method fetches an SKB from the neighbour `arp_queue` and calls the corresponding `solicit()` method to send it. In case of ARP, it will be `arp_solicit()`. It increments the neighbour `probes` counter and frees the packet.

int neigh_forced_gc(struct neigh_table *tbl)

This method is a synchronous garbage collection method. It removes neighbour entries that are not in the permanent state (NUD_PERMANENT) and whose reference count equals 1. The removal and cleanup of a neighbour is done by first setting the dead flag of the neighbour to be 1 and then calling the `neigh_cleanup_and_release()` method, which gets a neighbour object as a parameter. The `neigh_forced_gc()` method is invoked from the `neigh_alloc()` method under some conditions, as described in the "Creating and Freeing a Neighbour" section earlier in this chapter. The `neigh_forced_gc()` method returns 1 if at least one neighbour object was removed, and 0 otherwise.

void neigh_periodic_work(struct work_struct *work)

This method is the asynchronous garbage collector handler.

static void neigh_timer_handler(unsigned long arg)

This method is the per-neighbour periodic timer garbage collector handler.

struct neighbour *__neigh_lookup(struct neigh_table *tbl, const void *pkey, struct net_device *dev, int creat)

This method performs a lookup in the specified neighbouring table by the given key. If the `creat` parameter is 1, and the lookup fails, call the `neigh_create()` method to create a neighbour entry in the specified neighbouring table and return it.

neigh_hh_init(struct neighbour *n, struct dst_entry *dst)

This method initializes the L2 cache (`hh_cache` object) of the specified neighbour based on the specified routing cache entry.

void __init arp_init(void)

This method performs the setup for the ARP protocol: initialize the ARP table, register the `arp_rcv()` as a handler for receiving ARP packets, initialize `procfs` entries, register `sysctl` entries, and register the ARP `netdev` notifier callback, `arp_netdev_event()`.

int arp_rcv(struct sk_buff *skb, struct net_device *dev, struct packet_type *pt, struct net_device *orig_dev)

This method is the Rx handler for ARP packets (Ethernet packets with type 0x0806).

int arp_constructor(struct neighbour *neigh)

This method performs ARP neighbour initialization.

int arp_process(struct sk_buff *skb)

This method, invoked by the `arp_rcv()` method, handles the main processing of ARP requests and ARP responses.

void arp_solicit(struct neighbour *neigh, struct sk_buff *skb)

This method sends the solicitation request (ARPOP_REQUEST) after some checks and initializations, by calling the `arp_send()` method.

void arp_send(int type, int ptype, __be32 dest_ip, struct net_device *dev, __be32 src_ip, const unsigned char *dest_hw, const unsigned char *src_hw, const unsigned char *target_hw)

This method creates an ARP packet and initializes it with the specified parameters, by calling the `arp_create()` method, and sends it by calling the `arp_xmit()` method.

void arp_xmit(struct sk_buff *skb)

This method actually sends the packet by calling the NF_HOOK macro with `dev_queue_xmit()`.

struct arphdr *arp_hdr(const struct sk_buff *skb)

This method fetches the ARP header of the specified SKB.

int arp_mc_map(__be32 addr, u8 *haddr, struct net_device *dev, int dir)

This method translates an IPv4 address to L2 (link layer) address according to the network device type. When the device is an Ethernet device, for example, this is done with the `ip_eth_mc_map()` method; when the device is an Infiniband device, this is done with the `ip_ib_mc_map()` method.

static inline int arp_fwd_proxy(struct in_device *in_dev, struct net_device *dev, struct rtable *rt)

This method returns 1 if the specified device can use proxy ARP for the specified routing entry.

static inline int arp_fwd_pvlan(struct in_device *in_dev, struct net_device *dev, struct rtable *rt, __be32 sip, __be32 tip)

This method returns 1 if the specified device can use proxy ARP VLAN for the specified routing entry and specified IPv4 source and destination addresses.

int arp_netdev_event(struct notifier_block *this, unsigned long event, void *ptr)

This method is the ARP handler for `netdev` notification events.

int ndisc_netdev_event(struct notifier_block *this, unsigned long event, void *ptr)

This method is the NDISC handler for `netdev` notification events.

int ndisc_rcv(struct sk_buff *skb)

This method is the main NDISC handler for receiving one of the five types of solicitation packets.

static int neigh_blackhole(struct neighbour *neigh, struct sk_buff *skb)

This method discards the packet and returns –ENETDOWN error (network is down).

static void ndisc_recv_ns(struct sk_buff *skb) and static void ndisc_recv_na(struct sk_buff *skb)

These methods handle receiving Neighbour Solicitation and Neighbour Advertisement, respectively.

static void ndisc_recv_rs(struct sk_buff *skb) and static void ndisc_router_discovery(struct sk_buff *skb)

These methods handle receiving router solicitation and router advertisement, respectively.

int ndisc_mc_map(const struct in6_addr *addr, char *buf, struct net_device *dev, int dir)

This method translates an IPv4 address to a L2 (link layer) address according to the network device type. In Ethernet under IPv6, this is done by the `ipv6_eth_mc_map()` method.

int ndisc_constructor(struct neighbour *neigh)

This method performs NDISC neighbour initialization.

void ndisc_solicit(struct neighbour *neigh, struct sk_buff *skb)

This method sends the solicitation request after some checks and initializations, by calling the `ndisc_send_ns()` method.

int icmpv6_rcv(struct sk_buff *skb)

This method is a handler for receiving ICMPv6 messages.

bool ipv6_addr_any(const struct in6_addr *a)

This method returns 1 when the given IPv6 address is the unspecified address of all zeroes (IPV6_ADDR_ANY).

int inet_addr_onlink(struct in_device *in_dev, __be32 a, __be32 b)

This method checks whether the two specified addresses are on the same subnet.

Macros

Now, let's look at the macros.

IN_DEV_PROXY_ARP(in_dev)

This macro returns `true` if `/proc/sys/net/ipv4/conf/<netDevice>/proxy_arp` is set or if `/proc/sys/net/ipv4/conf/all/proxy_arp` is set, where `netDevice` is the network device associated with the specified `in_dev`.

IN_DEV_PROXY_ARP_PVLAN(in_dev)

This macro returns `true` if `/proc/sys/net/ipv4/conf/<netDevice>/proxy_arp_pvlan` is set, where `netDevice` is the network device associated with the specified `in_dev`.

IN_DEV_ARPFILTER(in_dev)

This macro returns `true` if `/proc/sys/net/ipv4/conf/<netDevice>/arp_filter` is set or if `/proc/sys/net/ipv4/conf/all/arp_filter` is set, where `netDevice` is the network device associated with the specified `in_dev`.

IN_DEV_ARP_ACCEPT(in_dev)

This macro returns `true` if `/proc/sys/net/ipv4/conf/<netDevice>/arp_accept` is set or if `/proc/sys/net/ipv4/conf/all/arp_accept` is set, where `netDevice` is the network device associated with the specified `in_dev`.

IN_DEV_ARP_ANNOUNCE(in_dev)

This macro returns the max value of `/proc/sys/net/ipv4/conf/<netDevice>/arp_announce` and `/proc/sys/net/ipv4/conf/all/arp_announce`, where `netDevice` is the network device associated with the specified `in_dev`.

IN_DEV_ARP_IGNORE(in_dev)

This macro returns the max value of `/proc/sys/net/ipv4/conf/<netDevice>/arp_ignore` and `/proc/sys/net/ipv4/conf/all/arp_ignore`, where `netDevice` is the network device associated with the specified `in_dev`.

IN_DEV_ARP_NOTIFY(in_dev)

This macro returns the max value of `/proc/sys/net/ipv4/conf/<netDevice>/arp_notify` and `/proc/sys/net/ipv4/conf/all/arp_notify`, where `netDevice` is the network device associated with the specified `in_dev`.

IN_DEV_SHARED_MEDIA(in_dev)

This macro returns `true` if `/proc/sys/net/ipv4/conf/<netDevice>/shared_media` is set or if `/proc/sys/net/ipv4/conf/all/shared_media` is set, where `netDevice` is the network device associated with the specified `in_dev`.

IN_DEV_ROUTE_LOCALNET(in_dev)

This macro returns `true` if `/proc/sys/net/ipv4/conf/<netDevice>/route_localnet` is set or if `/proc/sys/net/ipv4/conf/all/route_localnet` is set, where `netDevice` is the network device associated with the specified `in_dev`.

neigh_hold()

This macro increments the reference count of the specified neighbour.

The neigh_statistics Structure

The neigh_statistics structure is important for monitoring the neighbouring subsystem; as mentioned in the beginning of the chapter, both ARP and NDISC export this structure members via procfs (/proc/net/stat/arp_cache and /proc/net/stat/ndisc_cache, respectively). Following is a description of its members and pointing out where they are incremented:

```
struct neigh_statistics {
        unsigned long allocs;           /* number of allocated neighs     */
        unsigned long destroys;         /* number of destroyed neighs     */
        unsigned long hash_grows;       /* number of hash resizes         */
        unsigned long res_failed;       /* number of failed resolutions   */
        unsigned long lookups;          /* number of lookups              */
        unsigned long hits;             /* number of hits (among lookups) */
        unsigned long rcv_probes_mcast; /* number of received mcast ipv6  */
        unsigned long rcv_probes_ucast; /* number of received ucast ipv6  */
        unsigned long periodic_gc_runs; /* number of periodic GC runs     */
        unsigned long forced_gc_runs;   /* number of forced GC runs       */
        unsigned long unres_discards;   /* number of unresolved drops     */
};
```

Here is a description of the members of the neigh_statistics structure:

- allocs: The number of the allocated neighbours; incremented by the neigh_alloc() method.
- destroys: The number of the destroyed neighbours; incremented by the neigh_destroy() method.
- hash_grows: The number of times that hash resize was done; incremented by the neigh_hash_grow() method.
- res_failed: The number of failed resolutions; incremented by the neigh_invalidate() method.
- lookups: The number of neighbour lookups that were done; incremented by the neigh_lookup() method and by the neigh_lookup_nodev() method.
- hits: The number of hits when performing a neighbour lookup ; incremented by the neigh_lookup() method and by the neigh_lookup_nodev() method, when you have a hit.
- rcv_probes_mcast: The number of received multicast probes (IPv6 only); incremented by the ndisc_recv_ns() method.
- rcv_probes_ucast: The number of received unicast probes (IPv6 only); incremented by the ndisc_recv_ns() method.
- periodic_gc_runs: The number of periodic GC invocations; incremented by the neigh_periodic_work() method.

- `forced_gc_runs`: The number of forced GC invocations; incremented by the `neigh_forced_gc()` method.
- `unres_discards`: The number of unresolved drops; incremented by the `__neigh_event_send()` method when an unresolved packet is discarded.

Table

Here is the table that was covered.

Table 7-1. *Network Unreachability Detection States*

Linux	Symbol
NUD_INCOMPLETE	Address resolution is in progress and the link-layer address of the neighbour has not yet been determined. This means that a solicitation request was sent, and you are waiting for a solicitation reply or a timeout.
NUD_REACHABLE	The neighbour is known to have been reachable recently.
NUD_STALE	More than ReachableTime milliseconds have elapsed since the last positive confirmation that the forward path was functioning properly was received.
NUD_DELAY	The neighbour is no longer known to be reachable. Delay sending probes for a short while in order to give upper layer protocols a chance to provide reachability confirmation.
NUD_PROBE	The neighbour is no longer known to be reachable, and unicast Neighbour Solicitation probes are being sent to verify reachability.
NUD_FAILED	Set the neighbour to be unreachable. When you delete a neighbour, you set it to be in the NUD_FAILED state.