ARP Initialization

The *Address Resolution Protocol (ARP)* is responsible for mapping IP addresses to MAC addresses. An *ARP neighbor* is a host system or router that can be reached in a single "hop" and uses link layer (MAC) addressing instead of network layer (IP) addressing.

ARP data structures

The root of the ARP data structures is the *struct neigh_table,* defined in *include/net/neighbour.h*. Collectively the *struct neigh_table* and the structures to which it points are the internal realization of the *arp cache*. Each network layer protocol that uses ARP has an associated *neigh_table*.

```
139 struct neigh_table
140 {
141     struct neigh_table  *next;
142     int                 family;
143     int                 entry_size;
144     int                 key_len;
145     __u32       (*hash)(void *pkey, struct net_device *);
146     int         (*constructor)(struct neighbour *);
147     int         (*pconstructor)(struct pneigh_entry *);
148     void        (*pdestructor)(struct pneigh_entry *);
149     void        (*proxy_redo)(struct sk_buff *skb);
150     char            *id;
151     struct neigh_parms  parms;
152 /* HACK. gc_* shoul follow parms without a gap! */
153     int                 gc_interval;
154     int                 gc_thresh1;
155     int                 gc_thresh2;
156     int                 gc_thresh3;
157     unsigned long       last_flush;
158     struct timer_list   gc_timer;
159     struct timer_list   proxy_timer;
160     struct sk_buff_head proxy_queue;
161     int                 entries;
162     rwlock_t            lock;
163     unsigned long       last_rand;
164     struct neigh_parms  *parms_list;
165     kmem_cache_t        *kmem_cachep;
166     struct tasklet_struct gc_task;
167     struct neigh_statistics stats;
168     struct neighbour    *hash_buckets[NEIGH_HASHMASK+1];
169     struct pneigh_entry *phash_buckets[PNEIGH_HASHMASK+1];
170 };
171
```

Functions of structure elements:

next:	Used to link into a list of neighbour tables. "*neigh_tables"* points to first table of this list. It appears that in addition to IPV4 only DECNET and IPV6 register neigbor tables.
family:	The protocol family (PF_INET).
id:	Symbolic name of the table ("*arp_cache*")
hash:	Hash function used to map next hop IP address to a specific hash queue. For IP ARP, this is the function *arp_hash().*
entry_size:	Size of the *struct neighbour* + *4* (presumably for key length).
key_len:	Length of key(in bytes) used by hash function. Since the IP address is the key, the value is 4 for ARP.
constructor:	Initializes new instances of *struct neighbour.* There is a *struct neighbor* entity for each element in the ARP cache. For IP ARP, this is the function *arp_constructor().*
kmem_cache_p:	A pointer to a slab allocator cache of *struct neighbours*
hash_buckets:	*struct neighbour* hash queues. The hash and lookup key here is the next hop IP address.
phash_buckets:	*struct pneigh_entry* hash queues. These are (presumably) used in the proxy arp facility.
gc_thresh*	These values are used has high water marks for reducing the size of the ARP cache if it should grow too large.

There are 32 hash *struct neighbor* hash queues and 16 *struct pneigh* hash queues.

```
131 #define NEIGH_HASHMASK   0x1F
132 #define PNEIGH_HASHMASK  0xF
```

The IPv4 neighbor table

The neighbor table for the IPV4 ARP protocol is statically declared as follows:

```
164 struct neigh_table arp_tbl = {
165     family:         AF_INET,
166     entry_size:     sizeof(struct neighbour) + 4,
167     key_len:        4,
168     hash:           arp_hash,
169     constructor:    arp_constructor,
170     proxy_redo:     parp_redo,
171     id:             "arp_cache",
172     parms: {
173         tbl:                    &arp_tbl,
174         base_reachable_time:    30 * HZ,
175         retrans_time:           1 * HZ,
176         gc_staletime:           60 * HZ,
177         reachable_time:         30 * HZ,
178         delay_probe_time:       5 * HZ,
179         queue_len:              3,
180         ucast_probes:           3,
181         mcast_probes:           3,
182         anycast_delay:          1 * HZ,
183         proxy_delay:            (8 * HZ) / 10,
184         proxy_qlen:             64,
185         locktime:               1 * HZ,
186        },
187     gc_interval:    30 * HZ,
188     gc_thresh1:     128,
189     gc_thresh2:     512,
190     gc_thresh3:     1024,
191 };
```

The *entry_size* field is set to 4 more than it "needs to be" because of the way *struct neighbor* which is shown on the next page is defined. Its last field, *primary_key[0],* is declared as an array of 0 bytes. The extra 4 bytes of the *entry_size* ensure that when the stucture is dynamically allocated, space for the actual size of its *primary_key[0]* will be included.

The parms section defines some operational time-out triggers. In a standard x86 Linux system the clock ticks once every 10msec and HZ is equal to 100, the number of ticks per second.

```
4 #ifndef HZ
5 #define HZ 100
6 #endif
```

Thus Hz can be considered to mean seconds here.

The *neigh_parms* structure

The *struct neigh_parms* is defined in *include/net/neighbour.h*. Instances of, or pointers to this structure are contained in *neigh_table, neighbour,* and *in_device structures.*

```
53 struct neigh_parms
54 {
55     struct neigh_parms *next;
56     int   (*neigh_setup)(struct neighbour *);
57     struct neigh_table *tbl;
58     int   entries;
59     void  *priv;
60
61     void  *sysctl_table;
62
63     int   base_reachable_time;
64     int   retrans_time;
65     int   gc_staletime;
66     int   reachable_time;
67     int   delay_probe_time;
68
69     int   queue_len;
70     int   ucast_probes;
71     int   app_probes;
72     int   mcast_probes;
73     int   anycast_delay;
74     int   proxy_delay;
75     int   proxy_qlen;
76     int   locktime;
77 };
```

The *struct neighbour*

This structure defines the contents of a single arp cache element.

```
 87 struct neighbour
 88 {
 89     struct neighbour    *next;
 90     struct neigh_table  *tbl;
 91     struct neigh_parms  *parms;
 92     struct net_device   *dev;
 93     unsigned long       used;
 94     unsigned long       confirmed;
 95     unsigned long       updated;
 96     __u8                flags;
 97     __u8                nud_state;
 98     __u8                type;
 99     __u8                dead;
100     atomic_t            probes;
101     rwlock_t            lock;
102     unsigned char       ha[(MAX_ADDR_LEN+sizeof(unsigned
                            long)-1)&~(sizeof(unsigned long)-1)];
103     struct hh_cache     *hh;
104     atomic_t            refcnt;
105     int                 (*output)(struct sk_buff *skb);
106     struct sk_buff_head arp_queue;
107     struct timer_list   timer;
108     struct neigh_ops    *ops;
109     u8                  primary_key[0];
110 };
```

Functions of structure elements:

next:	Used to link the elements of a specific *hash_bucket*.
tbl:	Back pointer to the *neigh_table* that owns this structure.
parms:	Back pointer to the *parms* component of parent *neigh_table*
primary_key:	Place holder for unsigned 32-bit dest IP address, used by *hash* function. The actual space for the field is dynamically allocated.
ha:	Hardware (MAC) address of the remote connected network device.
hh_cache:	Pointer to the hardware header cache structure that is associated with the on-link destination node related to this arp cache element.
output:	A pointer to the function used to transmit the packet. This will point to *dev_queue_xmit()* when the arp cache entry is NUD_REACHABLE and will point to *neigh_resolve_output()* when it is not.
arp_queue:	A list of *sk_buffs* held because the state is presently *not* ARP_VALID
dev:	Points to the *net_device* structure associated with the interface with which this ARP cache entry is associated.
timer_list:	A kernel timer used for managing various time-out conditions
ops:	Table of function pointers from which (among other things) the value of *output* is taken.

The *hh_cache* structure

Hardware header cache elements contain the hardware header needed for the 1st hop made by an outgoing packet.

```
182 struct hh_cache
183 {
184     struct hh_cache *hh_next;     /* Next entry  */
185     atomic_t        hh_refcnt;    /* number of users   */
186     unsigned short  hh_type;      /* protocol id, ETH_P_IP
187                     *  NOTE:  For VLANs, this will be the
188                     *  encapuslated type. --BLG
189                     */
190     int             hh_len;       /* length of header      */
191     int             (*hh_output)(struct sk_buff *skb);
192     rwlock_t        hh_lock;
193 /* cached hardware header; allow for machine alignment     */
194     unsigned long   hh_data[16/sizeof(unsigned long)];
195 };
```

Functions of structure elements:

hh_next:	Link to next *hh_cache* structure.
hh_refcnt:	Reference count which controls deletion
hh_len:	Length of MAC layer header
hh_data:	Place holder for the hardware header itself.
hh_output:	A pointer to the *dev_queue_xmit()* or *neigh_resolve_output* function.

The *struct pneigh_entry*, presumably, describes a *Proxy* neighbour.

```
124 struct pneigh_entry
125 {
126         struct pneigh_entry     *next;
127         struct net_device       *dev;
128         u8                      key[0];
129 };
```

Neighbour operations

Each neighbour structure defines functions for a set of operations through the *neigh_ops* structure. This structure is filled in by its constructor which in turn is defined by its parent *neigh_table*.

```
112 struct neigh_ops
113 {
114     int  family;
115     void (*destructor)(struct neighbour *);
116     void (*solicit)(struct neighbour *, struct sk_buff*);
117     void (*error_report)(struct
                                neighbour *, struct sk_buff*);
118     int  (*output)(struct sk_buff*);
119     int  (*connected_output)(struct sk_buff*);
120     int  (*hh_output)(struct sk_buff*);
121     int  (*queue_xmit)(struct sk_buff*);
122 };
```

The *arp_constructor()* function sets the *neigh_ops* structure for a neighbour to any one of the following below based on the output device used to reach it. These are defined in net/ipv4/arp.c.

Generic *neigh_ops* structure.

```
126 static struct neigh_ops arp_generic_ops = {
127         family:                 AF_INET,
128         solicit:                arp_solicit,
129         error_report:           arp_error_report,
130         output:                 neigh_resolve_output,
131         connected_output:       neigh_connected_output,
132         hh_output:              dev_queue_xmit,
133         queue_xmit:             dev_queue_xmit,
134 };
```

The *neigh_ops* structure for devices *that require a hardware header*. This is the structure that will be used for Ethernet devices.

```
136 static struct neigh_ops arp_hh_ops = {
137         family:                 AF_INET,
138         solicit:                arp_solicit,
139         error_report:           arp_error_report,
140         output:                 neigh_resolve_output,
141         connected_output:       neigh_resolve_output,
142         hh_output:              dev_queue_xmit,
143         queue_xmit:             dev_queue_xmit,
144 };
```

The *neigh_ops* structure for neighbours that do not require ARP.

```
146 static struct neigh_ops arp_direct_ops = {
147         family:                 AF_INET,
148         output:                 dev_queue_xmit,
149         connected_output:       dev_queue_xmit,
150         hh_output:              dev_queue_xmit,
151         queue_xmit:             dev_queue_xmit,
152 };
```

The *neigh_ops* structure for device types that are broken.

```
154 struct neigh_ops arp_broken_ops = {
155         family:                 AF_INET,
156         solicit:                arp_solicit,
157         error_report:           arp_error_report,
158         output:                 neigh_compat_output,
159         connected_output:       neigh_compat_output,
160         hh_output:              dev_queue_xmit,
161         queue_xmit:             dev_queue_xmit,
162 };
```

The *arp_init()* function

Defined in net/ipv4/arp.c

Called by *inet_init();*

Responsibilities include:

Setting up the ARP cache.

Registering ARP packet type with kernel.

Creating a proc entry /proc/net/arp.

```
1193 void __init arp_init (void)
1194 {
1195     neigh_table_init(&arp_tbl);
1196
1197     dev_add_pack(&arp_packet_type);
1198
1199     proc_net_create ("arp", 0, arp_get_info);
1200
1201 #ifdef CONFIG_SYSCTL
1202     neigh_sysctl_register(NULL, &arp_tbl.parms,
        NET_IPV4, NET_IPV4_NEIGH, "ipv4");
1203 #endif
1204
```

Neighbor Table Initialization

Each major protocol family may provide its own address resolution service and neighbor table. At present IPV6 and DECNET provide their own services and IPV4 uses this generic ARP.

The *neigh_table _init()* function is defined in net/core/neighbour.c.

```
1114 void neigh_table_init(struct neigh_table *tbl)
1115 {
1116     unsigned long now = jiffies;
1117
```

Here the value of *reachable_time* is set to a random value uniformly distributed in:

[*base_reachable_time / 2, 3 x base_reachable_time*]

Recall that *base_reachable_time* is 30 seconds.

```
1118     tbl->parms.reachable_time =
              neigh_rand_reach_time(tbl->parms.
                                    base_reachable_time);
```

A cache named *arp_cache* is created. The *struct neighbour* objects will be allocated from this cache by the slab allocator. The value of *entry_size* has been previously set to *sizeof(struct neighbor) + 4*.

```
1120     if (tbl->kmem_cachep == NULL)
1121          tbl->kmem_cachep = kmem_cache_create(tbl->id,
1122                                (tbl->entry_size+15)&~15,
1123                                0, SLAB_HWCACHE_ALIGN,
1124                                 NULL, NULL);
1125
```

ARP uses kernel timers to drive exit routines used to check for time-out conditions. Each timer structure contains the following data elements:

```
16 struct timer_list {
17          struct    list_head list;
18          unsigned  long      expires;
19          unsigned  long      data;
20          void (*function)(unsigned long);
21 };
```

data:	An arbitrary value to be passed to the timer exit routine
function:	The address of the exit routine to be called
expires:	The time, in *jiffies*, at which the routine should be called.

The *init_timer()* function simply initializes the elements of the *timer_list* structure. Calling *add_timer()* arms the timer. Here the arbitrary data is a pointer to the *neigh_table* itself and the expiration is set to 30 * HZ + (~30 * HZ) = roughly 1 minute.

```
1129      init_timer(&tbl->gc_timer);
1130      tbl->lock = RW_LOCK_UNLOCKED;
1131      tbl->gc_timer.data = (unsigned long)tbl;
1132      tbl->gc_timer.function = neigh_periodic_timer;
1133      tbl->gc_timer.expires = now + tbl->gc_interval +
                                  tbl->parms.reachable_time;
1134      add_timer(&tbl->gc_timer);
1135
```

The proxy timer is created but not armed until a proxy arp element is established.

```
1136      init_timer(&tbl->proxy_timer);
1137      tbl->proxy_timer.data = (unsigned long)tbl;
1138      tbl->proxy_timer.function = neigh_proxy_process;
1139      skb_queue_head_init(&tbl->proxy_queue);
1140
1141      tbl->last_flush = now;
1142      tbl->last_rand = now + tbl->parms.reachable_time*20;
```

The initialized neighbour table (*arp_tbl)* is inserted into list of neighbour tables, pointed to by the global variable *neigh_tables*.

```
1143      write_lock(&neigh_tbl_lock);
1144      tbl->next = neigh_tables;
1145      neigh_tables = tbl;
1146      write_unlock(&neigh_tbl_lock);
1147 }
```

Registering the ARP packet type

After setting up the ARP cache, *arp_init()* must register the ARP packet type with the link layer. This is done via a call to *dev_add_pack()*.

```
1197     dev_add_pack(&arp_packet_type);
```

The *arp_packet_type* is statically declared as

```
1187 static struct packet_type arp_packet_type = {
1188     type:   constant_htons(ETH_P_ARP),
1189     func:   arp_rcv,
1190     data:   (void*) 1, /* understand shared skbs */
1191 };
```

The *arp_rcv()* is the packet handling function invoked on receiving an ARP packet. The parameters passed to it are shown below.

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

Creating /proc/net/arp entry

After registering ARP packet type, *arp_init()* creates a proc entry that displays the contents of ARP cache via *arp_get_info()*. *arp_get_info()* displays entries in *hash_buckets* and *phash_buckets*.

```
1199     proc_net_create ("arp", 0, arp_get_info);
1201 #ifdef CONFIG_SYSCTL
1202     neigh_sysctl_register(NULL, &arp_tbl.parms,
                           NET_IPV4, NET_IPV4_NEIGH, "ipv4");
1203 #endif
1204 }
```

In the following table, the last three entries are proxies:

```
/proc/net ==> cat arp
IP address       HW type     Flags       HW address            Mask     Device
192.168.2.4      0x1         0x2         00:00:77:97:C3:A5     *        lec0
192.168.2.5      0x1         0x2         00:00:77:88:A4:95     *        lec0
192.168.2.6      0x1         0x2         00:00:77:88:A1:15     *        lec0
192.168.2.35     0x1         0x2         00:50:DA:31:3F:4A     *        eth0
192.168.2.7      0x1         0x2         00:00:77:88:A5:A5     *        lec0
192.168.2.1      0x1         0x2         00:20:48:2E:00:EE     *        lec0
130.127.48.184   0x1         0xc         00:00:00:00:00:00     *        lec0
192.168.2.66     0x1         0xc         00:00:00:00:00:00     *        lec0
192.168.2.35     0x1         0xc         00:00:00:00:00:00     *        lec0
```