The IP - ARP Interface

Overview

There are *two distinct points* at which IP output processing interacts with ARP. The first occurs whenever a new route cache element is created by *ip_route_output_slow()*. At this time a new *struct neighbour* is created, initialized, and inserted in the ARP hash table.

Attachment of a *neighbour* structure to a route cache entry

The *ip_route_output_slow()* path is used to resolve routes that are not found in the route cache. After a route is successfully resolved, and a new route cache element is created, the *ip_route_output_slow()* function invokes the *rt_intern_hash()* function which is responsible for adding the new route to the hash queue.

```
        err = rt_intern_hash(hash, rth, rp);
```

```
static int rt_intern_hash(unsigned hash, struct rtable
                    *rt, struct rtable **rp)
```

If *rt_intern_hash()* succeeds in adding the new route to the cache it invokes *arp_bind_neighbour()* whose mission is to fill in the *rt->u.dst->neighbour* entry with a pointer to an ARP *struct neighbour*. This *neighbour* structure will *be permanently bound* to the route cache element.

```
        /* Try to bind route to arp only if it is output
             route or unicast forwarding path.
        */
        if (rt->rt_type == RTN_UNICAST || rt->key.iif== 0) {
             int err = arp_bind_neighbour(&rt->u.dst);
```

The *arp_bind_neighbour* function

The *arp_bind_neighbour()* function defined in *net/ipv4/arp.c* is invoked. This function tries to locate or create an entry in the ARP table for the destination address. Because many *different* routes may have the same first hop the relationship between *struct rtable* and *struct neighbour* is many to one.

The function depends upon *__neigh_lookup_errno()* to find an existing usable *struct neighbour* or to create a new one. The two parameters that comprise the lookup key are:

- a pointer to the next hop IP address and
- the outgoing *net_device*.

On exit, *dst->neighbour* will point to an initialized *neighbour* structure, *but address resolution will not have been performed if a new neighbour was created.*

The use of *dev* in the lookup key is important because MAC layer reachability is interface dependent.

```
int arp_bind_neighbour(struct dst_entry *dst)
{
    struct net_device *dev = dst->dev;
    struct neighbour *n = dst->neighbour;

    if (dev == NULL)
         return -EINVAL;
    if (n == NULL) {
         u32 nexthop = ((struct rtable*)dst)->rt_gateway;
         if (dev->flags&(IFF_LOOPBACK|IFF_POINTOPOINT))
              nexthop = 0;
         n = __neigh_lookup_errno(
#ifdef CONFIG_ATM_CLIP
              dev->type == ARPHRD_ATM ? &clip_tbl :
#endif
              &arp_tbl, &nexthop, dev);
         if (IS_ERR(n))
              return PTR_ERR(n);
         dst->neighbour = n;
    }
    return 0;
}
```

ARP neighbor lookup

The *neigh_lookup_errno()* function is defined in *include/net/neighbour.h.* The *neigh_table* pointer that is passed in here is the address of the statically allocated *arp_tbl.* This pointer is passed on through to lower level functions such as *neigh_lookup()* that do the real work.

The value of *pkey* is the IP address of the next hop gateway, but the combination of *pkey* and net device address must match for a lookup to succeed.

```
static inline struct neighbour *
    _neigh_lookup_errno(struct neigh_table *tbl,
                        const void *pkey,
                        struct net_device *dev)
{
    struct neighbour *n = neigh_lookup(tbl, pkey, dev);

    if (n)
         return n;

```

If *neigh_lookup()* doesn't find an entry *neigh_create()* will attempt to build one.

```
    return neigh_create(tbl, pkey, dev);
}
```

Neighbour lookup

The *neigh_lookup()* function defined in *net/core/neighbour.c* performs the actual lookup of neighbour structure from the input key. The *tbl->hash()* function pointer actually points to *arp_hash().*

```
struct neighbour *neigh_lookup(struct neigh_table *tbl,
                   const void *pkey, struct net_device *dev)
{
        struct neighbour *n;
        u32 hash_val;
        int key_len = tbl->key_len;
        hash_val = tbl->hash(pkey, dev);
```

Using the hash value from the *arp_hash* function as an index to the hash table, *neigh_lookup* tries to locate a neighbour entry whose *net_device* pointer matches the specified output device and whose key matches the IP address of the destination or gateway router (`nexthop = ((struct rtable*)dst)->rt_gateway`).

If an entry is found the *neigh_hold* macro increments its *refcnt* field. This reference is *not released* on return to *ip_route_output_slow().* This ensures that a *neighbour* will never be deleted out from under the *struct rtable.* If the requested entry is not found, the *for* loop will be exited with $n = 0$.

```

        read_lock_bh(&tbl->lock);
        for (n = tbl->hash_buckets[hash_val]; n;
                            n = n->next) {
             if (dev == n->dev &&
                 memcmp(n->primary_key, pkey,key_len) == 0) {
                 neigh_hold(n);
                 break;
            }
        }
        read_unlock_bh(&tbl->lock);
        return n;
}
```

The *neigh_hold()* function is defined as a macro in *include/net/neighbour.h*. It increments the reference count of the neighbour structure passed to it.

```
#define neigh_hold(n)   atomic_inc(&(n)->refcnt)
```

The *arp_hash* function

The hashing function for the ARP table, *arp_hash* is passed the next hop IP address and and outgoing *net_device* structure as parameters.

This function returns an index to the hash table of neighbour lists for storing or looking up the neighbour structure. It uses the input key (neighbour ip address) and the device interface index in computing the key.   The value of NEIGH_HASHMASK is 0x1f.

The *void *pkey* declaration "pretends" that the structure of the key is not constrained, but line 218 shows that is not exactly so.

```
214 static u32 arp_hash(const void *pkey, const struct
                                            net_device *dev)
215 {
216     u32 hash_val;
217
218     hash_val = *(u32*)pkey;
219     hash_val ^= (hash_val>>16);
220     hash_val ^= hash_val>>8;
221     hash_val ^= hash_val>>3;
222     hash_val = (hash_val^dev->ifindex)&NEIGH_HASHMASK;
223
224     return hash_val;
225 }
```

Neighbour Creation

When the desired neighbour is not found, it is the mission of the *neigh_create()* function, defined in net/core/neighbour.c, to create a new *neighbour* structure.

```
struct neighbour * neigh_create(struct neigh_table *tbl,
                   const void *pkey, struct net_device *dev)
{
    struct neighbour *n, *n1;
    u32 hash_val;
    int key_len = tbl->key_len;
    int error;

```

The *neigh_alloc()* function allocates the new neighbour structure and initializes a number of inportant fields including *parms, output,* and *state.*

```
    n = neigh_alloc(tbl);
```

On return to *neigh_create()* the new structure is further initialized. It's *primary_key* and *dev* fields are set to the value of parameters passed as input to the function.

```
    if (n == NULL)
         return ERR_PTR(-ENOBUFS);

    memcpy(n->primary_key, pkey, key_len);
    n->dev = dev;
```

The *dev_hold()* function increments the reference count of the device to reflect the fact that this neighbor structure now holds a pointer to it. It is decremented only when *neigh_release* is called to release this structure.

```
    dev_hold(dev);
```

As we have seen reference counting is a widely used and safe way to ensure the consistency of kernel data structures. It is, however crucial that the *holds* graph be *acyclic!*

Completing the initialization of the *neighbour* structure

The *neigh_alloc()* function which was called on the previous page performs generic initialization of the newly allocated *neighbour,* but IPV4 specific initialization is performed here.

The constructor field of the *arp_tbl* structure points to the *arp_constructor()* function, which is invoked here. In case of error, the neighbour structure is released.

```
/* Protocol specific setup. */
        if (tbl->constructor && error = tbl->constructor(n)) < 0)
        {
                neigh_release(n);
                return ERR_PTR(error);
        }
```

The *arp_tbl* structure, whose *parms* field is again referenced here does not define a *neigh_setup* function.

```
/* Device specific setup. */
        if (n->parms->neigh_setup &&
                        error = n->parms->neigh_setup(n)) < 0) {
                neigh_release(n);
                return ERR_PTR(error);
        }
```

The *confirmed* field is initialized to the present time in *jiffies* minus twice the *base_reachable_time* (which is set to 30 seconds in the *parms* structure of the *arp_tbl*). Time is warped backward here to ensure that the *nud_state* will not "accidentally" get set to *NUD_REACHABLE* during a subsequent timer interrupt.

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

Adding the neighbour to the hash chain

The hash value is computed and used to check if a neighbour structure with an identical key and device now exists. If one has been created via a race condition, then the new neighbour structure is released and the old one is returned after incrementing it's reference count.

```
    hash_val = tbl->hash(pkey, dev);

    write_lock_bh(&tbl->lock);
    for (n1 = tbl->hash_buckets[hash_val]; n1;
                                    n1 = n1->next) {
         if (dev == n1->dev && memcmp(n1->primary_key,
                    pkey, key_len)   == 0) {
              neigh_hold(n1);
              write_unlock_bh(&tbl->lock);
              neigh_release(n);
              return n1;
         }
    }
```

Next, the new neighbour structure is inserted at the head of the proper hash queue.

```
    n->next = tbl->hash_buckets[hash_val];
    tbl->hash_buckets[hash_val] = n;
```

Once, linked to the hash table, the dead field is reset to zero and it's reference count is incremented. Since both the hash queue and the *struct rtable* hold references there *must* be two distinct calls made to *neigh_hold()*.

```
    n->dead = 0;
    neigh_hold(n);
    write_unlock_bh(&tbl->lock);
    NEIGH_PRINTK2("neigh %p is created.\n", n);
    return n;
}
```

The *neigh_alloc* function

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

```
static struct neighbour *neigh_alloc(
                            struct neigh_table *tbl)
{
    struct neighbour *n;
    unsigned long now = jiffies;

```

If the number of entries in the table exceeds the *gc_thresh3* value (1024), or if the number of entries exceeds the *gc_thresh2* value (512) and the time since entries in the arp_cache were flushed exceeds 500 ticks (5 seconds), the *neigh_forced_gc()* routine defined in net/core/neighbour.c is invoked to shrink the table. This function removes old entries in the NUD_STALE state for which no-one holds a reference. If it is not successful in reducing the number of entries to fewer than 1024, NULL is returned indicating that *this allocation failed!*

```
    if (tbl->entries > tbl->gc_thresh3 ||
              tbl->entries > tbl->gc_thresh2 &&
              now - tbl->last_flush > 5*HZ)) {
         if (neigh_forced_gc(tbl) == 0 &&
                   tbl->entries > tbl->gc_thresh3)
              return NULL;
    }
```

With the number of entries in the table now less than 1024, the *neigh_alloc()* functions allocates a new neighbour structure from the cache.

```
    n = kmem_cache_alloc(tbl->kmem_cachep, SLAB_ATOMIC);
    if (n == NULL)
         return NULL;
```

Initialization of the *neighbour* structure

To ensure consistent state the *entire* *neighbour* structure is set to 0. The *entry_size* field in the *arp_tbl* was set to *sizeof(struct neighbour)* plus four bytes for storing the primary key as was described in the *arp_init* chapter. The remainder of this function initializes the new neighbour structure.

```
    memset(n, 0, tbl->entry_size);

    skb_queue_head_init(&n->arp_queue);
    n->lock = RW_LOCK_UNLOCKED;
    n->updated = n->used = now;
    n->nud_state = NUD_NONE;
    n->output = neigh_blackhole;
    n->parms = &tbl->parms;
```

The *neigh_timer_handler()* function is used to handle neighbour probe timeouts. The neighbour structure address is passed to this function as data.

```
    init_timer(&n->timer);
    n->timer.function = neigh_timer_handler;
    n->timer.data = (unsigned long)n;
    tbl->stats.allocs++;
    neigh_glbl_allocs++;
    tbl->entries++;
    n->tbl = tbl;
    atomic_set(&n->refcnt, 1);
```

The *dead* flag actually means "being created" here. It will be resent to 0 when the new entry is safely on the proper hash queue.

```
    n->dead = 1;
    return n;
}
```

The *arp_constructor* function

The *arp_constructor()* function defined in net/ipv4/arp.c is invoked from *neigh_create()* each time a *struct neighbor* is created.

```
227 static int arp_constructor(struct neighbour *neigh)
228 {
229     u32 addr = *(u32*)neigh->primary_key;
230     struct net_device *dev = neigh->dev;
231     struct in_device *in_dev = in_dev_get(dev);
```

If an *in_device* is not associated with the *net_device,* the *arp_constructor* function returns error. The address type is recovered by the *inet_addr_type()* function. If ARP parameters have already been associated with the *in_dev,* they are used instead of the generic parameters defined by *arp_tbl*. During inet device initialization, the **inetdev_init()* function calls *neigh_parms_alloc(dev, &arp_tbl)* which basically copies the *neigh_parms* from the *arp_tbl!*

```
232
233             if (in_dev == NULL)
234                     return -EINVAL;
```

The *type* value will distinguish *loopback, multicast, broadcast, and unicas*t.

```
235
236             neigh->type = inet_addr_type(addr);
237             if (in_dev->arp_parms)
238                     neigh->parms = in_dev->arp_parms;
```

The *in_dev_put()* decrements count of the *in_dev()* structure, and if there are no more references to it, *in_dev_finish_destroy* function defined in net/ipv4/devinet.c is called. Destruction can't possibly happen here because the counter was incremented in the call to *in_dev_get()* in line 231.

```
239
240             in_dev_put(in_dev);
```

neigh_ops selection

Ethernet devices always have a *hard_header()* function pointer. The function, *ether_setup(struct net_device *dev)* which is defined in drivers/net/net_init.c, sets this pointer as follows: *dev->hard_header = eth_header;* The *eth_header()* function is defined in net/ethernet/eth.c and its mission is to construct a hardware header within the *sk_buff.*

If the device doesn't need any hardware header, the neighbour state is set to NUD_NOARP and the *ops* structure for this neighbour is set to *arp_direct_ops*. The output function to be used for transmitting packets to this neighbour is set to *dev_queue_xmit*.

```
    if (dev->hard_header == NULL) {
         neigh->nud_state = NUD_NOARP;
         neigh->ops = &arp_direct_ops;
         neigh->output = neigh->ops->queue_xmit;
```

The device does have a hardware header. Most of this code is special case handling of odd-ball devices. An ethernet device should take the *default:* case.

```
    } else {
      /* Good devices (checked by reading
         texts,   but only Ethernet is
         tested)

         ARPHRD_ETHER: (ethernet, apfddi)
         ARPHRD_FDDI: (fddi)   */
         ARPHRD_IEEE802: (tr)
         ARPHRD_METRICOM: (strip)
         ARPHRD_ARCNET:
         etc. etc. etc.

```

If the device is one of the broken ones listed below, the ops field and the output function are appropriately initialised and arp_constructor returns else for good devices, we continue.

```
#if 1
                 /* So... these "amateur" devices are
                    hopeless. The only thing, that I can
                    say now:
                    It is very sad that we need to keep ugly
                    obsolete code to make them happy.

                    They should be moved to more reasonable
                    state, now they use rebuild_header
                    INSTEAD OF hard_start_xmit!!!
                    Besides that, they are sort of out of
                    date (a lot of redundant clones/copies,
                    useless in 2.1), I wonder why people
                    believe that they work.
                  */

         switch (dev->type) {
         default:
              break;
         case ARPHRD_ROSE:
#if defined(CONFIG_AX25) || defined(CONFIG_AX25_MODULE)
         case ARPHRD_AX25:
#if defined(CONFIG_NETROM) || defined(CONFIG_NETROM_MODULE)
         case ARPHRD_NETROM:
#endif
              neigh->ops = &arp_broken_ops;
              neigh->output = neigh->ops->output;
              return 0;
#endif
         ;}
#endif
```

Multicast, broadcast and loopback

If the neighbour address is a multicast address, it's state is set to NUD_NOARP. The *arp_mc_map()* function maps the neighbour multicast address to a multicast MAC type address. This address is entered in the neighbour structure's hardware address field as well.

```
        if (neigh->type == RTN_MULTICAST) {
            neigh->nud_state = NUD_NOARP;
            arp_mc_map(addr, neigh->ha, dev, 1);
```

For loopback devices and devices that do not need ARP, the state is set to NUD_NOARP and the hardware address from the device is copied to the neighbour structure's hardware address field.

```
        } else if (dev->flags&(IFF_NOARP| IFF_LOOPBACK)) {
            neigh->nud_state = NUD_NOARP;
            memcpy(neigh->ha, dev->dev_addr,dev->addr_len);
```

If the neighbour address type is broadcast, it's state is set to NUD_NOARP and the broadcast address of the device is set as the harware address of the neighbour.

```
        } else if (neigh->type == RTN_BROADCAST
                        || dev->flags&IFF_POINTOPOINT) {
            neigh->nud_state = NUD_NOARP;
            memcpy(neigh->ha, dev->broadcast,
                                  dev->addr_len);
        }
```

Setting up the *ops* and *output* pointers

For ethernet devices the *hard_header_cache* pointer is also set in the *net_init()* function: *dev->hard_header_cache = eth_header_cache;* Thus, the *neigh_ops* structure is set to point to *arp_hh_ops*.

```
    if (dev->hard_header_cache)
        neigh->ops = &arp_hh_ops;
    else
        neigh->ops = &arp_generic_ops;
```

If the neighbour state is one of NUD_VALID states (i.e. NUD_PERMANENT or NUD_NOARP or ....) , the output function is set to the *connected_output* member in it's ops structure. Otherwise it is set to the *output* member. For ethernet devices there is no difference as both point to the function *neigh_resolve_output()*.

```
    if (neigh->nud_state & NUD_VALID)
        neigh->output = neigh->ops->connected_output;
    else
        neigh->output = neigh->ops->output;
    }
    return 0;
}

static struct neigh_ops arp_hh_ops = {
        family:                 AF_INET,
        solicit:                arp_solicit,
        error_report:           arp_error_report,
        output:                 neigh_resolve_output,
        connected_output:       neigh_resolve_output,
        hh_output:              dev_queue_xmit,
        queue_xmit:             dev_queue_xmit,
};
```

ARP address resolution

Address resolution is triggered by the *ip_finish_output2()* function which was described in the *netfilter* section. At this point in the processing, the *skb->dst* pointer will point to a valid *dst_entry* element in the route cache. The code below is taken from the *ip_finish_output2()* function.

There are two mechanisms by which calls to the link layer may be made. If the *dst_entry* has an *hh_cache* pointer, then the *hh_cache* entry must contain both the hardware header itself and a pointer to an output function at the device layer. The *hh_output()* function is set to *dev_queue_xmit()* if the ARP cache element is in the NUD_CONNECTED state, If the *neighbour* structure transitions to the NUD_STALE state, the *neigh_suspect()* function will reset the *hh_output()* pointer to *neigh_resolve_output()*.

If there is no *hh* pointer in the *dst_entry*, the *neighbor* pointer that was established when the route cache entry was constructed will be used. This neighbor structure has an *output* function pointer which was set to *neigh->ops->output.* For ethernet devices, this function is *neigh_resolve_output()*. Otherwise (for a loopback, point to point, or virtual device) it set to invoke *dev_queue_xmit()* by the *arp_constructor()* function that is called when each neighbour structure was created.

```
     struct dst_entry *dst = skb->dst;
     struct hh_cache *hh = dst->hh;
 :
     if (hh) {
          read_lock_bh(&hh->hh_lock);
          memcpy(skb->data - 16, hh->hh_data, 16);
          read_unlock_bh(&hh->hh_lock);
          skb_push(skb, hh->hh_len);
          return hh->hh_output(skb);
     } else if (dst->neighbour)
          return dst->neighbour->output(skb);
```

If there is no hardware header structure and no neighbor structure available, then there is no way to send the packet and it must be dropped.

The *neigh_resolve_output* function

The neighbour hardware address resolver routine, *neigh_resolve_output(),* is defined in *net/core/neighbour.c.* This function is indirectly invoked by *ip_finish_output2()* as shown on the previous page when a cached hardware header is not available in the route destination entry or if the existing ARP cache element has become *stale.*

It's job is to resolve the next hop hardware address using *neigh_event_send()* routine. If *neigh_event_send()* returns success immediately, the hardware header is immediately copied to the *sk_buff,* and it is pushed on to the device.

```
/* Slow and careful. */

int neigh_resolve_output(struct sk_buff *skb)
{
        struct dst_entry *dst = skb->dst;
        struct neighbour *neigh;

        if (!dst || !(neigh = dst->neighbour))
                goto discard;
```

This step ensures that both the *nh.raw* pointer and the *data* pointer point to the start of the IP header and that *skb->len* reflects the number of bytes between *skb->tail* and the IP header. The *__skb_pull* macro adjusts the *skb->data* pointer by `skb->nh.raw - skb->data` and *skb->len* by the same amount.

```
        __skb_pull(skb, skb->nh.raw - skb->data);
```

Invoking *neigh_event_send*

The *neigh_event_send()* function is invoked here regardless of the state of the *neighbour.* It will actually send a probe packet only if the *neighbour* is in the NUD_NONE state. The function returns a NULL value if the *neighbour* structure is in one of the NUD_VALID states. In this case the *sk_buff* will be sent without further delay.

When a new *neighbour* structure has just been created, it will be in the NUD_NONE state, which is not in the NUD_VALID set, and *neigh_event_send()* will return 1. In this case the the *sk_buff* will be enqueued in the *arp_queue* of the *struct neighbour* where it is held until an ARP reply is successfully received by the *arp_rcv()* function. If no reply is received after several retries, the queue is purged.

```
    if (neigh_event_send(neigh, skb) == 0) {
        int err;
        struct net_device *dev = neigh->dev;
```

Neighbour state is NUD_VALID

If control reached this point *neigh_event_send()* returned 0. This indicates that the *neighbour* state is in the NUD_VALID state set. If the device has a *hard_header_cache()* function, and if the *dst_entry* structure doesn't have an *hh_cache* pointer, then the *neigh_hh_init()* function is invoked to initialize the *hh_cache* pointer in the *dst_entry*. This situation could possibly occur when a *new struct rtable is attached to an existing struct neighbour*. The value of *dst->ops->protocol* has been previously set to *ETH_P_IP* from the *ipv4_dst_ops* structure. In all cases the destination MAC address will be taken from *neigh->ha.*

```
        if (dev->hard_header_cache && dst->hh == NULL) {
             write_lock_bh(&neigh->lock);
             if (dst->hh == NULL)
                  neigh_hh_init(neigh, dst,
                                    dst->ops->protocol);
```

Constructing the hardware header

Next the device specific *hard_header()* function is called. Its mission is to construct the MAC header in the *kmalloc'd* portion of the *sk_buff* structure. For ethernet devices, a pointer to the *eth_header()* routine has been set in *dev->hard_header*. The *neigh->ha* field is used as *the destination hardware address.* This is safe because the state is NUD_VALID. The NULL value in the hardware source address field implies that the *source address should be copied from the net_device* structure.

```
             err = dev->hard_header(skb, dev,
                              ntohs(skb->protocol),
                              neigh->ha, NULL, skb->len);
             write_unlock_bh(&neigh->lock);
```

If the route destination already has a cached hardware header, the device specific *hard_header* function is directly invoked to construct the hardware header in the *sk_buff*. For ethernet devices, this function is *eth_header().*

```
        } else {
             read_lock_bh(&neigh->lock);
             err = dev->hard_header(skb, dev,
                             ntohs(skb->protocol),
                             neigh->ha, NULL, skb->len);
             read_unlock_bh(&neigh->lock);
        }
```

Passing the packet to dev_queue_xmit()

If there were no errors in setting the hardware header (The *dev->hard_header* function returns the length of the hardware header as successful return value), then the packet is queued for transmission.

```
            if (err >= 0)
                 return neigh->ops->queue_xmit(skb);
            kfree_skb(skb);
            return -EINVAL;
       }
```

If *neigh_event_send()* returned 1 a return is made here as well.

```
       return 0;

discard:
  NEIGH_PRINTK1("neigh_resolve_output: dst=%p
             neigh=%p\n", dst, dst ? dst->neighbour : NULL);
  kfree_skb(skb);
  return -EINVAL;
}
```

```
static struct neigh_ops arp_hh_ops = {
        family:                 AF_INET,
        solicit:                arp_solicit,
        error_report:           arp_error_report,
        output:                 neigh_resolve_output,
        connected_output:       neigh_resolve_output,
        hh_output:              dev_queue_xmit,
        queue_xmit:             dev_queue_xmit,
};
```

Constructing the *hh_cache* element

The *neigh_hh_init()* function is responsible for setting up the *hh_cache* pointer in the *dst_entry* structure. Recall that *neigh_hh_init()* is invoked if the *neighbour* is in a NUD_VALID state *but dst->hh_cache* is NULL.

```
static void neigh_hh_init(struct neighbour *n,
                      struct dst_entry *dst, u16 protocol)
{
    struct hh_cache *hh = NULL;
    struct net_device *dev = dst->dev;

```

The following loop attempts to find an *hh_cache* structure that is already linked to the neighbour structure and has the proper protocol type (ETH_P_IP). Under what conditions will multiple *hh_cache* elements be linked to a single *neighbour?* The only possible cause would be different network layer protocol types. Presumably, when a new *struct rtable* is attached to an existing *struct neighbour* that is in a NUD_VALID state, there will be an existing *hh_cache* structure.

```
    for (hh=n->hh; hh; hh = hh->hh_next)
         if (hh->hh_type == protocol)
              break;

```

If none was found, it is necessary to allocate a new one. Presumably this path will be taken when an ARP reply is received in the NUD_INCOMPLETE state.

```
    if (!hh && (hh = kmalloc(sizeof(*hh),
              GFP_ATOMIC)) != NULL) {
         memset(hh, 0, sizeof(struct hh_cache));
         hh->hh_lock = RW_LOCK_UNLOCKED;
         hh->hh_type = protocol;
         atomic_set(&hh->hh_refcnt, 0);
         hh->hh_next = NULL;
```

Initializing the new *hh_cache* structure

Normally this call will be to *eth_header_cache()*. It will fill in the *hh_cache* structure and return 0 if DIX framing is being used on the device and -1 if 802.3 framing is in use. The *eth_header_cache()* function takes the destination MAC address from the *n->ha* field of the *struct neighbour.* This would imply that this field must be filled in *before* *neigh_hh_init()* is invoked.

```
         if (dev->hard_header_cache(n, hh)) {
              kfree(hh);
              hh = NULL;
         } else {
              atomic_inc(&hh->hh_refcnt);
              hh->hh_next = n->hh;
              n->hh = hh;
              if (n->nud_state&NUD_CONNECTED)
                   hh->hh_output = n->ops->hh_output;
              else
                   hh->hh_output = n->ops->output;
        }
    }
```

Binding the *hh_cache* to the *dst_entry.*

The *hh_cache* structure was bound to the neigbours list above. Here it is also bound to the *dst_entry.*

```
    if (hh) {
         atomic_inc(&hh->hh_refcnt);
         dst->hh = hh;
    }
}
```

Initializing the *hh_cache* element

The DIX MAC header contains

source MAC address
dest MAC address
protocol type   (e.g. ETH_P_IP = 0x800)

```
int eth_header_cache(struct neighbour *neigh,
                               struct hh_cache *hh)
{
        unsigned short type = hh->hh_type;
        struct ethhdr *eth = (struct ethhdr*)
                                    (((u8*)hh->hh_data) + 2);
        struct net_device *dev = neigh->dev;

        if (type == __constant_htons(ETH_P_802_3))
              return -1;

        eth->h_proto = type;
        memcpy(eth->h_source, dev->dev_addr, dev->addr_len);
        memcpy(eth->h_dest, neigh->ha, dev->addr_len);
        hh->hh_len = ETH_HLEN;
        return 0;
}

```

The *neigh_event_send()* function

The *neigh_event_send* () function defined in include/net/neighbour.h is the generic wrapper used to send ARP requests. If the neighbour state is *not* one of NUD_CONNECTED states, the NUD_PROBE or the NUD_DELAY states then the *__neigh_event_send ()* function is called.

It will be seen that that an ARP request will be sent in only in the NUD_NULL state. Also *__neigh_event_send()* returns 1 in the NUD_INCOMPLETE state indicating that the packet cannot be presently sent and returns 0 in the NUD_STALE state indicating that it can be sent.

Normally *neigh_resolve_output()* would not have been called in the first place is the state is NUD_CONNECTED. But *neigh_resolve_output()* is called in NUD_DELAY and NUD_PROBE. But in these two states it *neigh_event_send()* simply returns 0.

```
static inline int neigh_event_send(
             struct neighbour *neigh, struct sk_buff *skb)
{
    neigh->used = jiffies;
    if  (!(neigh->nud_state &
              (NUD_CONNECTED|NUD_DELAY|NUD_PROBE)))
         return __neigh_event_send(neigh, skb);
    return 0;
}
```

Actions taken in the NUD_NONE, NUD_INCOMPLETE and NUD_STALE states

The *__neigh_event_send()* function defined in *net/core/neighbour.c* is called only in the NUD_NONE, NUD_INCOMPLETE, or NUD_STALE states. In NUD_NONE, it sends an ARP request and transitions to NUD_INCOMPLETE. In NUD_INCOMPLETE it puts the *skbuff* on the ARP queue. In NUD_STALE it transitions to NUD_DELAY.

```
int __neigh_event_send(struct neighbour *neigh,
                              struct sk_buff *skb)
{
        write_lock_bh(&neigh->lock);
        if  (!(neigh->nud_state &
                  (NUD_CONNECTED|NUD_DELAY|NUD_PROBE))) {
```

NUD_INCOMPLETE implies that a request is presently pending. Thus the only way to enter this block appears to be in the NUD_NONE state which is the initial state for new *struct neighbours.*

The multiple levels of nesting and *long* if constructs are contrary to the kernel coding guidelines and make this mess almost incomprehensible.

```
             if (!(neigh->nud_state & (NUD_STALE |
                       NUD_INCOMPLETE))) {
```

The NUD_NULL handler

The initial values of *mcast_probes, ucast_probes,* and *app_probes* are set to 3, 3, and 0 respectively in *arp_tbl.* When a *neighbour* is thought to be in a NUD_VALID state, ARP requests should be unicast, not broadcast. However, if a system has changed MAC addresses, it will not respond to unicast ARP requests. Therefore, if no response is received in unicast mode, ARP must fall back and issue broadcast requests. The *total* number of requests that may be issued, and after which ARP will give up, is the sum of the unicast and broadcast requests.

However, in the NUD_NULL state, it is *not possible to issue unicast requests.* So here *neigh_probes* is set to *ucast->probes* leaving the total number of remaining probes equal *mcast->probes.* The value *app_probes* is meaningful only if the ARP deamon is in use. The use of *mcast* is misleading. ARP packets are either unicast or broadcast in reality.

```
                if (neigh->parms->mcast_probes +
                        neigh->parms->app_probes) {
                        atomic_set(&neigh->probes,
                             neigh->parms->ucast_probes);
                        neigh->nud_state = NUD_INCOMPLETE;
                        neigh_hold(neigh);
```

The timer is set to expire in 1 second and the *arp_solicit()* function is invoked indirectly. The timer function here is *neigh_timer_handler.* Then the *arp_solicit()* function is invoked to multicast the ARP packet. After the call to *arp_solicit, neigh->probes* is set to 4.

```
                        neigh->timer.expires = jiffies +
                                  neigh->parms->retrans_time;
                        add_timer(&neigh->timer);
                        write_unlock_bh(&neigh->lock);
                        neigh->ops->solicit(neigh, skb);
```

The probes field in the neighbour structure is incremented after sending the arp packet giving it a value of 4.

```
                        atomic_inc(&neigh->probes);
                        write_lock_bh(&neigh->lock);
```

Sum of *mcast_probes* and *app_probes* is zero

```
      } else {
```

If the sum of *mcast_probes* and *app_probes* is zero, it likely indicates that the neighbour doesn't support probes. The neighbour state is set to NUD_FAILED, the *sk_buff* is freed and failure is returned to the caller.

```
          neigh->nud_state = NUD_FAILED;
          write_unlock_bh(&neigh->lock);

          if (skb)
             kfree_skb(skb);
          return 1;
      }
}
```

Adding the *sk_buff* to the *arp_queue*

If the neighbour state is NUD_INCOMPLETE , a request (possibly generated by this call to this function) is presently pending . If the input *sk_buff* pointer is valid, the *sk_buff* is queued at the end of the *arp_queue* of the *struct neighbour* and failure is returned. If the length of the *arp queue* is greater than the value set in the *struct neighbour* (set to 3 in *arp_tbl*), the *first* *sk_buff* on the list is dropped.

```
        if (neigh->nud_state == NUD_INCOMPLETE) {
                if (skb) {
                        if (skb_queue_len(&neigh->arp_queue) >=
                              neigh->parms->queue_len) {
                                struct sk_buff *buff;
                                buff = neigh->arp_queue.next;
                                __skb_unlink(buff,
                                             &neigh->arp_queue);
                                kfree_skb(buff);
                        }
                        __skb_queue_tail(&neigh->arp_queue, skb);
                }

                write_unlock_bh(&neigh->lock);
                return 1;
        }
```

The *NUD_STALE* state

If the state is NUD_STALE , the timer expiration time is set to the *delay_probe_time* field (which is set to 5*HZ (seconds) in the *arp_tbl)* and a transition to the NUD_DELAY state occurs. However, in this case no ARP request is sent yet.

```
        if (neigh->nud_state == NUD_STALE) {
            NEIGH_PRINTK2("neigh %p is delayed.\n", neigh);
            neigh_hold(neigh);
            neigh->nud_state = NUD_DELAY;
            neigh->timer.expires = jiffies +
                neigh->parms->delay_probe_time;
            add_timer(&neigh->timer);
        }
    }
```

A value of 0 is returned indicating that the *struct neighbour* is in a valid state and the packet may be queued for transmission.

```
    write_unlock_bh(&neigh->lock);
    return 0;
}
```

The *arp_solicit()* function

The *arp_solicit* defined in net/ipv4/arp.c sends an ARP request.

```
static void arp_solicit(struct neighbour *neigh,
                                       struct sk_buff *skb)
{
    u32 saddr;
    u8  *dst_ha = NULL;
    struct net_device *dev = neigh->dev;
```

The variable *target* holds the supposed neighbour's IP address. The variable *neigh->probes* was initialized to `neigh->parms->ucast_probes` when this function is called from the NUD_INCOMPLETE state and 0 when called from the NUD_PROBE state.

```
    u32 target = *(u32*)neigh->primary_key;
    int probes = atomic_read(&neigh->probes);
```

If the address of an *sk_buff* was passed in, the *struct neigbour* will be in the NUD_INCOMPLETE state and the *sk_buff* will be on the ARP queue. If the IP source address in the *sk_buff* is of type RTN_LOCAL, it is used as the source IP address in the ARP request. Otherwise a source address of type RT_SCOPE_LINK is selected from the device's IP address list.

```
    if (skb && inet_addr_type(skb->nh.iph->saddr)==
                                            RTN_LOCAL)
         saddr = skb->nh.iph->saddr;
    else
         saddr = inet_select_addr(dev, target,
                                       RT_SCOPE_LINK);
```

The unicast/broadcast decision

If the value of *probes* which was initialized to the *ucast_probes* parameter is now less than the *ucast_probes* parameter, the *dst_ha* pointer is set to the start of hardware address array of the neighbour structure. What is occuring here is related to the refreshing of a NUD_STALE entry. In that case *neigh_probes* is initialized to zero a unicast probe will be used until the total number of probes exceeds the allowable number of unicast probes. When that happens, *arp_solicit* will revert to broadcast probes as it should.

```
    if ((probes -= neigh->parms->ucast_probes) < 0) {
         if (!(neigh->nud_state & NUD_VALID))
              printk(KERN_DEBUG "trying to ucast
                             probe in NUD_INVALID\n");
         dst_ha = neigh->ha;
         read_lock_bh(&neigh->lock);
```

If ARPD is configured in the kernel and the neighbour structure's probe field is less than the *app_probes* parameter (which is 0 unless the ARPD is enabled), then the *neigh_app_ns* routine is invoked to send a message to the user-space arp daemon and return back to the caller.

```
    } else if ((probes -= neigh->parms->app_probes)< 0){
#ifdef CONFIG_ARPD
         neigh_app_ns(neigh);
#endif
         return;
    }
```

The *arp_send* ()routine is used to send the arp request to the neighbour or network.

```
    arp_send(ARPOP_REQUEST, ETH_P_ARP, target, dev,
                   saddr, dst_ha, dev->dev_addr, NULL);
    if (dst_ha)
         read_unlock_bh(&neigh->lock);
}
```

ARP packet structures

ARP packets consist of the protocol independent header shown in blue followed by a protocol dependent pair of hardware and protocol (IP) addresses.

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

The *type* parameter whose values are shown here. It will become the *ar_op* field in the *arphdr*.

```
#define ARPOP_REQUEST 1  /* ARP request                  */
#define ARPOP_REPLY 2    /* ARP reply                    */
```

The ARP hardware type and protocol type are set here based on the *type* of the *net_device* associated with the request. For Ethernet it will be `ARPHRD_ETHER`.

```
#define ARPHRD_ETHER 1    /* Ethernet 10Mbps              */
#define ARPHRD_EETHER 2   /* Experimental Ethernet        */
#define ARPHRD_AX25 3     /* AX.25 Level 2                */
```

The value of the *ar_pro* field is the protocol number which will 0x0800. The packet type in the MAC header will be be *0x806* for Ethernet based ARP.

```
#define ETH_P_IP  0x0800  /* Internet Protocol packet   */
#define ETH_P_X25 0x0805  /* CCITT X.25                 */
#define ETH_P_ARP 0x0806  /* Address Resolution packet  */
```

The *arp_send()* function

The *arp_send()* function, defined in ipv4/arp.c constructs and sends both ARP requests and ARP responses.

```

void arp_send(int type, int ptype, u32 dest_ip,
              struct net_device *dev, u32 src_ip,
              unsigned char *dest_hw, unsigned char
              *src_hw, unsigned char *target_hw)
{
```

If **dest_hw != *target_hw* this must be a refresh request for a proxy ARP relationship or possibly a case in which a host owns more than one interface. If *dest_hw* is NULL then it is necessary to do a hardware level broadcast of the packet.

```
    struct sk_buff *skb;
    struct arphdr *arp;
    unsigned char *arp_ptr;
```

If the specified device does not support ARP, an immediate return is made to the caller.

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

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

Buffer allocation

The *sk_buff* allocated for the ARP packet must hold the data *struct arphdr* , two copies of the MAC address *dev->addr_len,* two copies of the IP address (the constant 4) and the device hardware header length. GFP_ATOMIC allocation must be used because this routine may be called from timer (and possibly network) softirqs.

```
/*
 *      Allocate a buffer
 */

    skb = alloc_skb(sizeof(struct arphdr)+
                   2*(dev->addr_len+4) +
                   dev->hard_header_len + 15, GFP_ATOMIC);
    if (skb == NULL)
         return;
```

Space is reserved for the device hardware header length at the head of the *sk_buff* using the *skb_reserve* routine. *skb_put* allocates space for the ARP header, source and destination addresses in the buffer (ARP packet data) and returns the starting address of the allocated space.

```
    skb_reserve(skb, (dev->hard_header_len + 15)&~15);
    skb->nh.raw = skb->data;
    arp = (struct arphdr *) skb_put(skb,
         sizeof(struct  arphdr) + 2*(dev->addr_len+4));
```

MAC layer address selection

The device and protocol field of the *sk_buff* are initialized. If the hardware source address is not specified in the input parameter, the *source address of the device is used*. If the destination hardware address is not specifed, *the broadcast address of the device must be used*.

```
    skb->dev = dev;
    skb->protocol = __constant_htons (ETH_P_ARP);
    if (src_hw == NULL)
         src_hw = dev->dev_addr;
    if (dest_hw == NULL)
         dest_hw = dev->broadcast;
```

ARP packet construction

For ethernet devices, the *eth_header()* function is invoked here to fill in the hardware header in the *sk_buff*.

```
/*
 *      Fill the device header for the ARP frame
 */
   if (dev->hard_header && dev->hard_header(skb,
                  dev,ptype,dest_hw,src_hw,skb->len) < 0)
        goto out;
```

The ARP hardware type and protocol type are set here based on the *type* of the *net_device* associated with the request. For Ethernet it will be `ARPHRD_ETHER`.

```
#define ARPHRD_ETHER 1    /* Ethernet 10Mbps              */
#define ARPHRD_EETHER 2   /* Experimental Ethernet        */
#define ARPHRD_AX25 3     /* AX.25 Level 2                */
```

```
/*
 * Fill out the arp protocol part.
 *
 * The arp hardware type should match the device type,
   except for FDDI, which (according to RFC
   should always equal 1 (Ethernet).
 */
 /*
   Exceptions everywhere. AX.25 uses the AX.25
   PID value not the DIX code for the
   protocol. Make these device structure fields.
        */
   switch (dev->type) {
   default:
        arp->ar_hrd = htons(dev->type);
        arp->ar_pro = __constant_htons(ETH_P_IP);
        break;
```

```
#if defined(CONFIG_AX25) || defined(CONFIG_AX25_MODULE)
        case ARPHRD_AX25:
                arp->ar_hrd =
                           __constant_htons(ARPHRD_AX25);
                arp->ar_pro = __constant_htons(AX25_P_IP);
                break;


#if defined(CONFIG_NETROM) || defined(CONFIG_NETROM_MODULE)
        case ARPHRD_NETROM:
                arp->ar_hrd =
                        __constant_htons(ARPHRD_NETROM);
                arp->ar_pro = __constant_htons(AX25_P_IP);
                break;
#endif
#endif

#ifdef CONFIG_FDDI
        case ARPHRD_FDDI:
                arp->ar_hrd =
                         __constant_htons(ARPHRD_ETHER);
                arp->ar_pro = __constant_htons(ETH_P_IP);
                break;
#endif
#ifdef CONFIG_TR
        case ARPHRD_IEEE802_TR:
                arp->ar_hrd =
                         __constant_htons(ARPHRD_IEEE802);
                arp->ar_pro = __constant_htons(ETH_P_IP);
                break;
#endif
        }
```

The ARP hardware address length is set equal to the device address length and the protocol address length to four. The operation field (1 = ARP request, 2 = ARP response) is set equal to the input arp packet type parameter.

```
    arp->ar_hln = dev->addr_len;
    arp->ar_pln = 4;
    arp->ar_op = htons(type);
```

The *arp_ptr* points to the first byte after the generic arp header. The network/protocol specific ARP fields are stored from here. The source network hardware address, source protocol address, destination hardware address (for ARP responses) and destination protocol address are stored in that order.

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

    memcpy(arp_ptr, src_hw, dev->addr_len);
    arp_ptr+=dev->addr_len;
    memcpy(arp_ptr, &src_ip,4);
    arp_ptr+=4;
    if (target_hw != NULL)
         memcpy(arp_ptr, target_hw, dev->addr_len);
    else
         memset(arp_ptr, 0, dev->addr_len);
    arp_ptr+=dev->addr_len;
    memcpy(arp_ptr, &dest_ip, 4);
```

Finally *dev_queue_xmit()* is called to send the ARP packet.

```
    dev_queue_xmit(skb);
    return;

out:
    kfree_skb(skb);
}
```

The *eth_header()* function

```
int eth_header(struct sk_buff *skb, struct net_device *dev,
        unsigned short type,
        void *daddr, void *saddr, unsigned len)
{
    struct ethhdr *eth = (struct ethhdr *
                        skb_push(skb,ETH_HLEN);

    /*
     *      Set the protocol type. For a packet of type
             ETH_P_802_3 we put the length
     *      in here instead.
            It is up to the 802.2 layer to carry
            protocol information.
     */

    if(type!=ETH_P_802_3)
       eth->h_proto = htons(type);
    else
       eth->h_proto = htons(len);

    /*
     *      Set the source hardware address.
     */

    if(saddr)
       memcpy(eth->h_source,saddr,dev->addr_len);
    else
       memcpy(eth->h_source,dev->dev_addr,dev->addr_len);

     /*
      *   loopback-device should never use this function...
      */

     if (dev->flags & (IFF_LOOPBACK|IFF_NOARP))
     {
        memset(eth->h_dest, 0, dev->addr_len);
        return(dev->hard_header_len);
     }

     if(daddr)
     {
        memcpy(eth->h_dest,daddr,dev->addr_len);
        return dev->hard_header_len;
     }

     return -dev->hard_header_len;
}
```