ARP Receive

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; /* format of hardware address */
    unsigned short ar_pro; /* format 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]; /* send 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

};
```

```
09:05:57.478352 arp who-has jmw9 tell jmw7
0x0000       0001 0800 0604 0001 00b0 d0e9 0f5b c0a8    .............[..
0x0010       0221 0000 0000 0000 c0a8 022c              ..............

09:05:57.478376 arp reply jmw9 is-at 0:9:6b:e3:7e:a2
0x0000       0001 0800 0604 0002 0009 6be3 7ea2 c0a8    ..........k.~...
0x0010       022c 00b0 d0e9 0f5b c0a8 0221              .,.....[...!
```

The *arp_rcv()* function

The *arp_rcv()* function defined in *net/ipv4/arp.c* is the arp packet handler invoked by *net_rx_action()* when an ARP packet is received. In the following, the pointer, *arp,* references the ARP header structure and *arp_ptr* refers to the data consisting of the two MAC and two IP addresses.

```
/*
 *      Receive an arp request by the device layer.
 */

int arp_rcv(struct sk_buff *skb, struct net_device *dev,
                                               struct packet_type *pt)
{
    struct arphdr *arp = skb->nh.arph;
    unsigned char *arp_ptr= (unsigned char *)(arp+1);
    struct rtable *rt;
    unsigned char *sha, *tha;
    u32 sip, tip;
    u16 dev_type = dev->type;
    int addr_type;
    struct in_device *in_dev = in_dev_get(dev);
    struct neighbour *n;
```

Validating the ARP packet

The following validity checks are performed on received packets.

- An *in_device* structure must be associated with the device on which the packet was received.
- The hardware header length specified in the arp header must match that of the device
- The protocol address length must be 4, the length of an IP address.
- The device must support ARP.
- The *pkt_type* must not indicate this packet is a loopback or destined for another host.

```
if (in_dev == NULL ||
     arp->ar_hln != dev->addr_len    ||
     dev->flags & IFF_NOARP ||
     skb->pkt_type == PACKET_OTHERHOST ||
     skb->pkt_type == PACKET_LOOPBACK ||
     arp->ar_pln != 4)
     goto out;
```

If the *sk_buff* is shared, then it is cloned by the *skb_share_check()* function.

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

If the *sk_buff* is non-linear, it is linearized by the *skb_linearize()* function and *arp* and *arp_ptr* are reset to refer to the new data.

```
if (skb_is_nonlinear(skb)) {
     if (skb_linearize(skb, GFP_ATOMIC) != 0)
          goto freeskb;
     arp = skb->nh.arph;
     arp_ptr= (unsigned char *)(arp+1);
}
```

The type of device on which the arp packet arrived and the arp hardware address type should be the same and must be either ARPHRD_ETHER or ARPHRD_IEEE802. Similarly, the protocol type field of the arp packet should be ETH_P_IP.

```
    switch (dev_type) {
    default:
         if (arp->ar_pro !=  __constant_htons(ETH_P_IP))
              goto out;
         if (htons(dev_type) != arp->ar_hrd)
              goto out;
         break;


#ifdef CONFIG_NET_ETHERNET
    case ARPHRD_ETHER:
         /*
         ETHERNET devices will accept ARP hardware types of
         either (Ethernet) or 6 (IEEE 802.2).
         */
         if (arp->ar_hrd!=__constant_htons(ARPHRD_ETHER)&&
             arp->ar_hrd !=
                        __constant_htons(ARPHRD_IEEE802))
              goto out;
         if (arp->ar_pro !=  __constant_htons(ETH_P_IP))
              goto out;
         break;
#endif
```

A number of similar protocol dependent blocks (Token Ring, etc...) follow here.

```
    :
    }
```

Validation continues with the message type.

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

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

At this point the packet header is thought to be valid, and data fields in the ARP packet are retrieved to local variables. *sha* is *sender_hardware_address* and *tip* is *target_ip* address, etc. Note that *sha* and *tha* are pointers, but *sip* and *tip* are values.

```
/*
 *      Extract fields
 */

    sha=arp_ptr;
    arp_ptr += dev->addr_len;
    memcpy(&sip, arp_ptr, 4);
    arp_ptr += 4;
    tha=arp_ptr;
    arp_ptr += dev->addr_len;
    memcpy(&tip, arp_ptr, 4);
```

Bad requests for loopback and multicast addresses are dropped.

```
/*
        Check for bad requests for 127.x.x.x and requests   for
        multicast addresses.  If this is one such, delete it.
 */
        if (LOOPBACK(tip) || MULTICAST(tip))
             goto out;

```

If the packet arrived on a device of type ARPHRD_DLCI (frame relay DLCI ??), then the source hardware address of the packet is reset to broadcast address of the device ??

```
/*
        Special case: We must set Frame Relay source Q.922
        address
 */
        if (dev_type == ARPHRD_DLCI)
             sha = dev->broadcast;
```

Processing of validated packets

Actual processing of the packet begins here.

- For ARP responses it is necessary to update an existing *neighbour* structure.
- For ARP requests it is necessary to create or update a *neighbour* structure and send the reply.

Duplicate address detection

If the source IP address is NULL, and the packet is an ARP request, and the target ip address is of type RTN_LOCAL indicating that it is owned by this machine, then this is an IPv4 duplicate address detection packet. An ARP reply is sent immediately and no updating of the *neighbour* structures occurs.

```
/* Special case: IPv4 duplicate address detection
   packet (RFC2131) */
if (sip == 0) {
     if (arp->ar_op ==
               __constant_htons(ARPOP_REQUEST) &&
               inet_addr_type(tip) == RTN_LOCAL)
          arp_send(ARPOP_REPLY,ETH_P_ARP,tip,dev,
               tip,sha,dev->dev_addr,dev->dev_addr);
          goto out;
}
```

Processesing of ARP request packets

If the incoming ARP packet is an ARP request then, *ip_route_input()* is invoked. The objective of this is to determine if this host owns *tip*. The *ip_route_input()* function returns NULL if the packet is routeable and this host owns *tip* if the route type is RTN_LOCAL.

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

             rt = (struct rtable*)skb->dst;
             addr_type = rt->rt_type;
```

Processing of ARP requests for this host

If the route cache entry is of type RTN_LOCAL (i.e. the packet is for local delivery), then the packet is an ARP request for this host. The *neigh_event_ns()* function updates the ARP cache by creating a neighbour structure if necessary and caching the hardware address of the neighbour that initiated the ARP request.

```
        if (addr_type == RTN_LOCAL) {
            n = neigh_event_ns(&arp_tbl, sha, &sip, dev);
```

neigh_event_ns returns the updated neighbour structure on success and returns NULL on error.

```
            if (n) {
                int dont_send = 0;
```

If theARP filter option is enabled for IPV4 or in the *in_device* structure and the *arp_filter()* function validates the request. It return true when the ARP request is invalid.

```
                if(IN_DEV_ARPFILTER(in_dev))
                    dont_send |=
                        arp_filter(sip,tip,dev);
```

If *dont_send* remains false, an ARP reply is sent to the requesting neighbour. Parameters to ARP send are the *target ip, source ip, target hardware adddress, source hardware address.*

```
                if (!dont_send)
                        arp_send(ARPOP_REPLY,
                          ETH_P_ARP,sip,dev,tip,sha,
                          dev->dev_addr,sha);
                    neigh_release(n);
            }
```

This concludes the processing of ARP requests destined for this host. A jump is taken to the exit point, *out.*

```
              goto out;
```

ARP requests for other hosts

If route type of the ARP request packet was not local and if forwarding is enabled on the input device, then a check is made to see if this request requires a proxy arp reply.

```
        } else if (IN_DEV_FORWARD(in_dev)) {
```

If this is an ARP request for one of the neighbours for which we are acting as a proxy then one of the following conditions should hold true:

- RTCF_DNAT (destination NAT) flag is set in the route cache entry indicating that the packet destination address must be translated. For an ARP request, this indicates that intended destination is a neighbour "behind" this host.
- The address type of the next route is RTN_UNICAST *and* the device for the next hop is different from the device the packet arrived on and either proxy ARP is supported by the device or a proxy neighbour structure of the target host behind this host is already present in the cache.

If one of the above conditions is true, the hardware address of the ARP request source host is added to the neighbour cache by the *neigh_event_ns* routine.

```
                if ((rt->rt_flags&RTCF_DNAT) ||
                          addr_type == RTN_UNICAST  &&
                          rt->u.dst.dev != dev &&
                          (IN_DEV_PROXY_ARP(in_dev) ||
                          pneigh_lookup(&arp_tbl, &tip,
                          dev, 0)))) {
                     n = neigh_event_ns(&arp_tbl,
                                         sha, &sip, dev);
```

The *neigh_lookup()* function called by *neigh_event_ns()* above increments the reference count of the structure. Here, *neigh_release* is called to decrement the reference count after it has been used and updated above.

```
                     if (n)
                          neigh_release(n);
```

Proxy ARP replies

The *proxy_delay* parameter in the *arp_tbl* set to (8 * HZ) / 10, so by default all proxy ARP replies are delayed. The ARP reply is sent without delay only if any one of the following conditions is true. Otherwise it is queued bythe *pneigh_enqueue* routine.

- If the *stamp.tv_sec* field in the *sk_buff* has been reset to zero by *pneigh_enqueue* i.e. this *sk_buff* has been queued for a specific period of time. (Note: All incoming packets were time-stamped (i.e. *do_gettimeofday(&skb->stamp)*) back in *netif_rx* routine.) While it is true that *pneigh_enqueue()* does zero the time stamp and put the packet on the proxy_queue, it remains unclear how control could reach line 776 after proceding down that path.

- If the *sk_buff* packet type is PACKET_HOST or if the *proxy_delay* field in the *arp_parms* structure of *in_device* equals zero.

```
776                       if (skb->stamp.tv_sec == 0
                                    || 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);
780                       } else {
                               pneigh_enqueue(&arp_tbl,
                               in_dev->arp_parms, skb);
782                            in_dev_put(in_dev);
783                            return 0;
784                       }
785                       goto out;
786                  }
787             }
788        }
```

Handling ARP responses

Before an ARP request is sent, a *neighbour* structure must be created. Thus the *neigh_lookup()* function is called with the *creat* flag set to NULL indicating that a new *neighbour* should *not* be created if the lookup fails. If the lookup should fail, this packet is an unsolicited ARP response.

```
/* Update our ARP tables */
n = __neigh_lookup(&arp_tbl, &sip, dev, 0);
```

Handling of unsolicited ARP responses

Unsolicited ARP responses are not accepted unless CONFIG_IP_ACCEPT_UNSOLICITED_ARP is defined. If the neighbour lookup failed, and if the packet is an ARP reply with the IP source address of type RTN_UNICAST, then this is an unsolicited ARP reply. In this case *__neigh_lookup* is invoked a second time but with the *creat* flag set to create a new neighbour structure.

```
#ifdef CONFIG_IP_ACCEPT_UNSOLICITED_ARP
    /*
        Unsolicited ARP is not accepted by default.
        It is possible, that this option should be
        enabled for some devices (strip is candidate)
    */
    if (n == NULL &&
            arp->ar_op == __constant_htons(ARPOP_REPLY) &&
            inet_addr_type(sip) == RTN_UNICAST)
        n = __neigh_lookup(&arp_tbl, &sip, dev, -1);
#endif
```

Handling of solicited (and acceptable unsolicitied) ARP responses

If the neighbour lookup or creation is successful, the default new state of the neighbour structure in the cache is NUD_REACHABLE.

```
if (n) {
     int state = NUD_REACHABLE;
     int override = 0;

/*
     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.
 */
```

If the last update time of the neighbour structure is greater than the *locktime* parameter (set to 1*Hz in *arp_tbl*) of the neighbour, the *override* flag is set to true. The *override* flag permits a new hardware address to replace an existing one.

```
     if (jiffies - n->updated >= n->parms->locktime)
        override = 1;
```

If the ARP packet is not an ARP reply (how could control reach here in that case???) or if the packet type is not PACKET_HOST (unicast packet destined to this host) then the default state of the neighbour is reset to NUD_STALE .

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

The call to *neigh_update() generally* sets the state to NUD_REACHABLE if it's a direct ARP reply and to NUD_STALE if it is not. The call to *neigh_release()* decrements the reference count.

```
             neigh_update(n, sha, state, override, 1);
             neigh_release(n);
        }

out:
        if (in_dev)
             in_dev_put(in_dev);
freeskb:
        kfree_skb(skb);
out_of_mem:
        return 0;
}
```

ARP filters

IN_DEV_ARPFILTER has been defined in *include/linux/inetdevice.h*. The *ipv4_devconf* structure holds various IPv4 configuration values. A static variable of this structure named *ipv4_devconf* is declared in *net/ipv4/devinet.c* and initialized with the default values. By default *arp_filter* is turned off.

```
#define IN_DEV_ARPFILTER(in_dev) (ipv4_devconf.arp_filter
                              || (in_dev)->cnf.arp_filter)
```

```
struct ipv4_devconf
{
        int     accept_redirects;
        int     send_redirects;
        int     secure_redirects;
        int     shared_media;
        int     accept_source_route;
        int     rp_filter;
        int     proxy_arp;
        int     bootp_relay;
        int     log_martians;
        int     forwarding;
        int     mc_forwarding;
        int     tag;
        int     arp_filter;
        void    *sysctl;
};
```

```
struct ipv4_devconf ipv4_devconf = { 1, 1, 1, 1, 0, };
```

Note: These values can be configured using the old sysctl command interface or the present proc file system interface. These configuration values are rooted in the */proc/sys/net/ipv4/conf* directory.

The *arp_filter* function

The *arp_filter* function defined in *net/ipv4/arp.c* rejects the packet

- when a return route for the reply cannot be determined and
- when an output route is available but the output device is different from the device, the arp request arrived on.

In the case of a rejection, the neighbour structure is released and the packet is dropped in the *out* block of *arp_rcv().*

```
static int arp_filter(__u32 sip, __u32 tip,
                                 struct net_device *dev)
{
    struct rtable *rt;
    int flag = 0;
  /*unsigned long now; */

    if (ip_route_output(&rt, sip, tip, 0, 0) < 0)
         return 1;
    if (rt->u.dst.dev != dev) {
         NET_INC_STATS_BH(ArpFilter);
         flag = 1;
    }
    ip_rt_put(rt);
    return flag;
}
```

Updating ARP Cache entries

The *neigh_event_ns()* function defined in net/core/neighbour.c is called when ARP *requests* are received. It attempts to locate a *neighbour* structure with key equal to the source address of the ARP request packet. If successful, *neigh_update()* updates the structure using link layer address in the ARP packet. Note that *neigh_event_ns* sets the *neighbour* state to NUD_STALE, as it is not called in response to a direct ARP reply from the neighbour.

```
struct neighbour * neigh_event_ns(struct neigh_table *tbl,
                                  u8 *lladdr, void *saddr,
                                  struct net_device *dev)
{
        struct neighbour *neigh;

        neigh = __neigh_lookup(tbl, saddr, dev,
                               lladdr || !dev->addr_len);
        if (neigh)
                neigh_update(neigh, lladdr, NUD_STALE, 1, 1);
        return neigh;
}
```

The *neigh_update* function

The *neigh_update()* function is defined in *net/core/neighbour.c*. The input parameters are described in the comment block below. The parameter *lladdr* refers to the Link Layer or MAC address.

```
/* Generic update routine.
       -- lladdr is new lladdr or NULL, if it is not supplied.
       -- new is new state.
       -- override==1 allows to override existing lladdr,
          if it is different.
       -- arp==0 means that the change is administrative (i.e
            not generated by the arp protocol.
       Caller MUST hold reference count on the entry.
 */

int neigh_update(struct neighbour *neigh, const u8 *lladdr,
                                          u8 new, int override, int arp)
{
       u8 old;
       int err;
       int notify = 0;
       struct net_device *dev = neigh->dev;

       write_lock_bh(&neigh->lock);
       old = neigh->nud_state;

       err = -EPERM;
```

If the present neighbour state is either NUD_NOARP or NUD_PERMANENT, then it should not be changed regardless of what the caller might think!

```
       if (arp && (old&(NUD_NOARP|NUD_PERMANENT)))
            goto out;
```

New state not VALID

If the *new* state is *not* in the NUD_VALID set *{NUD_REACHABLE, NUD_PROBE, NUD_STALE, NUD_DELAY, NUD_PERMANENT, NUD_NOARP}* , then any timer that was set up before is deleted. It appears that the only way this can occur is when *neigh_delete()* sets the new state to *NUD_FAILED*.

```
if (!(new & NUD_VALID)) {
     neigh_del_timer(neigh);
```

New state not VALID and old state REACHABLE

If the *old* state was in the NUD_CONNECTED set *{NUD_REACHABLE, NUD_PERMANENT, NUD_NOARP}* and the new state is not in the NUD_VALID set, *neigh_suspect()* is called to update the output function pointers so that the sending of an ARP request will be triggered. The *notify* field is used to communicate the new state to the user-space ARP daemon.

```
     if (old & NUD_CONNECTED)
          neigh_suspect(neigh);
     neigh->nud_state = new;
     err = 0;
     notify = old & NUD_VALID;
     goto out;
}
```

New state *is* VALID

Devices not using link layer addresses

If the address length of the device is zero i.e. the device doesn't need an address, the *lladdr* field is set to point to the neighbour structure's hardware address field which presumably contains *nothing*.

```
/* Compare new lladdr with cached one */
 if (dev->addr_len == 0) {
/* First case: device needs no address. */
      lladdr = neigh->ha;
```

Device requires link layer address and one is specified

If the old state of the neighbour cache entry is valid then both the new hardware address and the cached hardware address are compared. If both are equal, then the *lladdr* pointer is reset to the existing the hardware address.. what does this accomplish? If they are not the same and the *override* flag is false, then processing is aborted.

```
} else if (lladdr) {
/* The second case: if something is already  cached
     and a new address is proposed:
     - compare new & old
     - if they are different, check override flag
 */
     if (old & NUD_VALID) {
          if (memcmp(lladdr, neigh->ha,
                                  dev->addr_len) == 0)
               lladdr = neigh->ha;
          else if (!override)
               goto out;
     }
```

Device requires link layer address but it is not specified

```
    } else {
    /* No address is supplied; if we know
       something, use it, otherwise discard the request.
    */
         err = -EINVAL;
```

If no new hardware address is supplied and the old state is not VALID there is nothing more that can be done.   However, if the state *is* valid the present link level address is used.

```
         if (!(old & NUD_VALID))
              goto out;
         lladdr = neigh->ha;
    }
```

Recovering the *old* state, part II.

The *neigh_sync()* function is called to determine the current state of the neighbor. This function effects the transitions between NUD_REACHABLE and NUD_STALE based upon whether or not the entry has last been *confirmed* within the *base_reachable_time* interval.

```
    neigh_sync(neigh);
    old = neigh->nud_state;
```

If the new state is one of the NUD_CONNECTED states (i.e. NUD_REACHABLE or NUD_NOARP or NUD_PERMANENT), then the confirmed time is updated. This appears to be the *only place neigh->confirmed* gets updated.

```
    if (new & NUD_CONNECTED)
         neigh->confirmed = jiffies;
    neigh->updated = jiffies;
```

If the present state of the neighbour is in the NUD_VALID set { NUD_REACHABLE, NUD_PROBE, NUD_STALE} and there is no change in the neighbour hardware address then if either of the following are true, the state is not updated.

- Both the proposed new state and the current state are equal
- The proposed new state is NUD_STALE and the current state is one of NUD_CONNECTED states. (Note: The value of *old* was set *after* the call to *neigh_sync()*. Thus this condition ensures that connected neighbour entries are not overridden when the input parameter is NUD_STALE).

```
   /* If entry was valid and address is not changed,
      do not change entry state, if new one is STALE.
   */
    err = 0;
    if (old & NUD_VALID) {
         if (lladdr == neigh->ha)
              if (new == old ||(new == NUD_STALE &&
                   (old & NUD_CONNECTED)))
                    goto out;
    }
```

Updating the neighbour state

Any pending timer is deleted and the new state is assigned to the neighbour.

```
        neigh_del_timer(neigh);
        neigh->nud_state = new;
```

If there is a change in the hardware address of the neighbour, the new address is copied to the neighbour and all cached hardware headers of the neighbour are updated by the *neigh_update_hhs()* function. This is the reason for the obscure reset of lladdr that was noted earlier.

```
        if (lladdr != neigh->ha) {
             memcpy(&neigh->ha, lladdr, dev->addr_len);
             neigh_update_hhs(neigh);
```

If the new state is not one of the NUD_CONNECTED states, then the *confirmed* ticks field is reset back by twice the *base_reachable_time*.

```
             if (!(new&NUD_CONNECTED))
                  neigh->confirmed = jiffies -
                   (neigh->parms->base_reachable_time<<1);
```

If user space ARP daemon is configured, the notify flag is set to enable the kernel to notify it later.

```
#ifdef CONFIG_ARPD
             notify = 1;
#endif
        }
```

If the *state* has not changed there is nothing more to be done. If the state is now in the NUD_CONNECTED set *neigh_connect()* is called to setup the fast transmit path. Otherwise the slow path is setup by *neigh_suspect()*.

```
        if (new == old)
             goto out;
        if (new&NUD_CONNECTED)
             neigh_connect(neigh);
        else
             neigh_suspect(neigh);
```

Draining the *arp_queue*

If the old state was not one of the NUD_VALID states and the new state *is* one of the NUD_VALID states, there may be *sk_buffs* awaiting output on the neighbour's *arp_queue* which can now be successfully transmitted. These *sk_buffs* are dequeued and queue for transmission.

```
    if (!(old & NUD_VALID)) {
         struct sk_buff *skb;

    /* Again: avoid dead loop if something went wrong */

         while (neigh->nud_state & NUD_VALID &&
              (skb= __skb_dequeue(&neigh->arp_queue))
                    != NULL) {
              struct neighbour *n1 = neigh;
              write_unlock_bh(&neigh->lock);
          /*  On shaper/eql
              skb->dst->neighbour != neigh :( */
              if (skb->dst && skb->dst->neighbour)
                   n1 = skb->dst->neighbour;
```

The output function of the neighbour is called here to push the packet on to the device after setting up the hardware header. And the arp_queue is purged after dequeuing them above, in case anything went wrong.

```
              n1->output(skb);
              write_lock_bh(&neigh->lock);
         }
         skb_queue_purge(&neigh->arp_queue);
    }

out:
        write_unlock_bh(&neigh->lock);
#ifdef CONFIG_ARPD
        if (notify && neigh->parms->app_probes)
                neigh_app_notify(neigh);
#endif
        return err;
}
```

The *neigh_synch()* function -

The *neigh_synch()* function updates the value of *nud_state* based upon the *confirmed* and *reachable* times.

```
static void neigh_sync(struct neighbour *n)
{
    unsigned long now = jiffies;
    u8 state = n->nud_state;

    ASSERT_WL(n);
    if (state & (NUD_NOARP|NUD_PERMANENT))
         return;
    if (state & NUD_REACHABLE) {
         if (now-n->confirmed > n->parms->reachable_time){
              n->nud_state = NUD_STALE;
              neigh_suspect(n);
          }
    } else if (state & NUD_VALID) {
         if (now-n->confirmed < n->parms->reachable_time) {
              neigh_del_timer(n);
              n->nud_state = NUD_REACHABLE;
              neigh_connect(n);
         }
    }
}
```