ARP Timer Management

There are two classes of timer associated with ARP processing. The `neigh_periodic_timer()` function is the handler for the `gc_timer` element of the `neigh_table` structure. It drives the process of expiring and refreshing all of the `neighbour` structures (ARP cache elements) owned by the `neigh_table`.

Each `neighbour` structure also has a private timer, `neigh->timer`, that is used to detect timeouts of ARP requests associated with that specific entry. The handler for all neighbour specific events is `neigh_timer_handler()`.

Periodic Timer Initialization

Recall that the `neigh_table_init()` function initialized the `neigh_periodic_timer` routine. On UP systems, this is implemented as a timer handler and on SMP systems this is scheduled as a tasklet by a timer routine. This function periodically checks the ARP cache, updating various fields and deleting any entries matching specific criteria.

```c
1114 void neigh_table_init(struct neigh_table *tbl)  
1115 {  
1116   unsigned long now = jiffies;  
1117   tbl->parms.reachable_time = neigh_rand_reach_time  
1118     (tbl->parms.base_reachable_time);  
```

On SMP systems, the `neigh_periodic_timer` routine is executed as a tasklet and the routine name is set by the `SMP_TIMER_NAME` macro.

```c
1126 #ifdef CONFIG_SMP  
1127   tasklet_init(&tbl->gc_task,  
1128     SMP_TIMER_NAME(neigh_periodic_timer),  
1129     (unsigned long)tbl);  
1130 #endif
```
On UP Systems, the `neigh_periodic_timer()` routine is invoked after the expiration of `gc_timer`. On SMP systems, it is run as a tasklet.

```
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);

1145  neigh_tables = tbl;
1146  write_unlock(&neigh_tbl_lock);
1147  }
```
Periodic timer handling

The SMP_TIMER_NAME macro is defined in include/linux/interrupt.h.

    #ifdef CONFIG_SMP
    #define SMP_TIMER_NAME(name) name##__thr
    #else /* CONFIG_SMP */
    #define SMP_TIMER_NAME(name) name
    #endif /* CONFIG_SMP */

The SMP_TIMER_NAME macro on SMP systems yields the name of the tasklet and this tasklet is scheduled by neigh_periodic_timer. On UP systems this macro yields neigh_periodic_timer, which is invoked by gc_timer. On SMP systems, neigh_periodic_timer when invoked by gc_timer schedules the SMP_TIMER_NAME(neigh_periodic_timer) function.

    #ifdef CONFIG_SMP
    static void neigh_periodic_timer(unsigned long arg)
    {
        struct neigh_table *tbl = (struct neigh_table*)arg;
        tasklet_schedule(&tbl->gc_task);
    }
    #endif

    static void SMP_TIMER_NAME(neigh_periodic_timer)
    (unsigned long arg)
    {
        struct neigh_table *tbl = (struct neigh_table*)arg;
        unsigned long now = jiffies;
        int i;
        write_lock(&tbl->lock);

        if (now - tbl->last_rand > 300*HZ) {
            struct neigh_parms *p;
            tbl->last_rand = now;
            for (p=&tbl->parms; p; p = p->next)
                p->reachable_time = neigh_rand_reach_time(
                                          p->base_reachable_time);
        }
    }

The first step is to run through the list of all neigh_parms structures randomizing the reachable times so as to avoid synchronized expiration of the neighbour specific timers. It is not clear that each neighbour actually has a private neigh_parms structure.
Processing the entire ARP cache

The function then inspects each *neighbour* entry in the ARP cache performing the following operations:

- Updating the the *used* time  
- Identifying neighbour structures that should be deleted  
- Identifying neighbour structures that should be marked *NUD_STALE*

One iteration of the outer for loop occurs for each hash chain and one iteration of the inner loop occurs for each *neighbour* on the chain being processed.

```c
for (i=0; i <= NEIGH_HASHMASK; i++) {
    struct neighbour *n, **np;
    np = &tbl->hash_buckets[i];
    while ((n = *np) != NULL) {
        unsigned state;
        write_lock(&n->lock);
        state = n->nud_state;
        if((state&(NUD_PERMANENT |NUD_IN_TIMER)) {  
            write_unlock(&n->lock);
            goto next_elt;
        }
        if ((long)(n->used - n->confirmed) < 0)
            n->used = n->confirmed;
    }
}
```

**States for which no processing is required**

No processing is needed for permanent cache entries and for the NUD_INCOMPLETE, NUD_DELAY, and NUD_PROBE states that consitute the NUD_IN_TIMER state.

```c
state = n->nud_state;
if((state&(NUD_PERMANENT |NUD_IN_TIMER)) {  
    write_unlock(&n->lock);
    goto next_elt;
}
```

If the last use time is less than the last confirmed time, the last use time is advanced to the last confirmed time.

```c
if (((long)(n->used - n->confirmed) < 0)
    n->used = n->confirmed;
```
Removal of FAILED or EXPIRED neighbours

If \( n->refcount \) is equal to 1, then this hash queue holds the only reference to the neighbour structure. In that case, if the state is NUD_FAILED or the time of last use exceeds the stale time threshold, \( gc_staletime \) (which defaults to 60 seconds), the neighbour structure is removed from the hash queue and deleted.

```c
587       if (atomic_read(&n->refcnt) == 1 &&
588              (state == NUD_FAILED ||
589              now - n->used >
590              n->parms->gc_staletime)) {
591          *np = n->next;
592          n->dead = 1;
593          write_unlock(&n->lock);
594          neigh_release(n);
595          continue;
596          }
```

The NUD_REACHABLE to NUD_STALE transition

If the state of the neighbour is NUD_REACHABLE and the last confirmed direct ARP reply from the neighbour was received approximately 30 seconds ago, then the entry state is set to NUD_STALE and slow output path is set by neigh_suspect routine.

```c
596       if (n->nud_state & NUD_REACHABLE &&
597              now - n->confirmed >
598              n->parms->reachable_time) {
599               n->nud_state = NUD_STALE;
600               neigh_suspect(n);
601               write_unlock(&n->lock);
602
603 next_elt:
604       np = &n->next;
605       }
```

On completion of the loop the timer is re-armed.

```c
608       mod_timer(&tbl->gc_timer, now + tbl->gc_interval);
609       write_unlock(&tbl->lock);
610   }
```
The neighbour specific timer handler

The `neigh_timer_handler()` function, defined in `net/core/neighbour.c`, handles neighbour specific ARP request timeouts.

```c
/* Called when a timer expires for a neighbour entry. */
static void neigh_timer_handler(unsigned long arg)
{
    unsigned long now = jiffies;
    struct neighbour *neigh = (struct neighbour*)arg;
    unsigned state;
    int notify = 0;
    write_lock(&neigh->lock);
    state = neigh->nud_state;

    if (!(state & NUD_IN_TIMER)) {
        ifndef CONFIG_SMP
            printk("neigh: timer & !nud_in_timer\n");
        endif
        goto out;
    }

    if ((state & NUD_VALID) &&
        now - neigh->confirmed < neigh->parms->reachable_time) {
        neigh->nud_state = NUD_REACHABLE;
        NEIGH_PRINTK2("neigh %p is still alive.\n", neigh);
        neigh_connect(neigh);
        goto out;
    }
}
```

Activation of this handler guarantees the neighbour entry state must be one of the NUD_IN_TIMER states (NUD_INCOMPLETE, NUD_PROBE or NUD_DELAY). So if its not something is badly amiss.

If the current neighbour state is VALID and if the the time since the last confirmed reply from the neighbour is LESS than the reachable time parameter, then the neighbour state is set to NUD_REACHABLE and the fast path output option is established. This could possibly happen if the response from the neighbour was received by the time the timer expired??

```c
if ((state & NUD_VALID) &&
    now - neigh->confirmed < neigh->parms->reachable_time) {
    neigh->nud_state = NUD_REACHABLE;
    NEIGH_PRINTK2("neigh %p is still alive.\n", neigh);
    neigh_connect(neigh);
    goto out;
}
```
Timeout in the NUD_DELAY state

If the current neighbour state is NUD_DELAY, it is set to NUD_PROBE and the probes field of the neighbour is reset to zero. This will trigger an ARP request later in this function.

```c
if (state == NUD_DELAY) {
    NEIGH_PRINTK2("neigh %p is probed.\n", neigh);
    neigh->nud_state = NUD_PROBE;
    atomic_set(&neigh->probes, 0);
}
```

Failed ARP requests

The `neigh_max_probes()` function returns the sum of `app_probes`, `mcast_probes` and `ucast_probes` parameters of the neighbour structure. In the case of `arp_tbl`, this sum is 6. If the current number of probes reaches this value, the neighbour state is set to NUD_FAILED, the `arp_queue` is purged.

```c
if (atomic_read(&neigh->probes) >= neigh_max_probes(neigh)) {
    struct sk_buff *skb;
    neigh->nud_state = NUD_FAILED;
    notify = 1;
    neigh->tbl->stats.res_failed++;
    NEIGH_PRINTK2("neigh %p is failed.\n",   neigh);

    /* It is very thin place.
       report_unreachable is very complicated routine.
       Particularly, it can hit the same neighbour entry!
       So that, we try to be accurate and avoid dead loop.
       --ANK
    */
    while(neigh->nud_state==NUD_FAILED &&
          (skb=__skb_dequeue(&neigh->arp_queue)) != NULL) {
        write_unlock(&neigh->lock);
        neigh->ops->error_report(neigh,skb);
        write_lock(&neigh->lock);
    }
    skb_queue_purge(&neigh->arp_queue);
    goto out;
}
```
Sending or retrying ARP requests

If control reaches this point, then either the neighbor just entered the NUD_PROBE state or an ARP request timed-out, but the maximum number of retries has not been reached. The timer is re-armed and the retry is initiated by calling the `arp_solicit` function with the address of first `sk_buff` on the `arp_queue` list of the neighbor (which will be NULL if the `neighbour` has passed through the the STALET>DELAY->PROBE transitions).

683   neigh->timer.expires = now + neigh->parms->retrans_time;
684   add_timer(&neigh->timer);
685   write_unlock(&neigh->lock);
686
687   neigh->ops->solicit(neigh, skb_peek(&neigh->arp_queue));
688   atomic_inc(&neigh->probes);
689   return;
690
691 out:
692   write_unlock(&neigh->lock);
693 #ifdef CONFIG_ARPD
694   if (notify && neigh->parms->app_probes)
695     neigh_app_notify(neigh);
696 #endif
697   neigh_release(neigh);
698 }
The `neigh_suspect()` and `neigh_connect` functions

The `neigh_suspect` and `neigh_connect` functions respectively disable and enable the fast path output function for a neighbour. The fast path output function is `dev_queue_xmit` and the slow output function is `neigh_resolve_output`. All the cached hardware headers are also updated with the new output function.

476 /* Neighbour state is suspicious; disable fast path. */
477          Called with write_locked neigh.
478 * /
479 static void neigh_suspect(struct neighbour *neigh) {
480             struct hh_cache *hh;
481     NEIGH_PRINTK2("neigh %p is suspecteded.\n", neigh);
482     ASSERT_WL(neigh);
483     neigh->output = neigh->ops->output;
484     for (hh = neigh->hh; hh; hh = hh->hh_next)
485         hh->hh_output = neigh->ops->output;
486 }

The `neigh->ops->connected_output` is actually `neigh_resolve_output`. Only `hh_output` links directly to `dev_queue_xmit`.

495 /* Neighbour state is OK; enable fast path. */
496          Called with write_locked neigh.
497 * /
498 static void neigh_connect(struct neighbour *neigh) {
499             struct hh_cache *hh;
500     NEIGH_PRINTK2("neigh %p is connected.\n", neigh);
501     ASSERT_WL(neigh);
502     neigh->output = neigh->ops->connected_output;
503     for (hh = neigh->hh; hh; hh = hh->hh_next)
504         hh->hh_output = neigh->ops->hh_output;
505 }

9
Handling deferred proxy ARP requests

For reasons that are not well understood, responses to proxy ARP requests are not issued immediately but delayed for a random timeout in the range of \([0, 0.8]\) seconds. The `pneigh_enqueue()` function, defined in `net/core/neighbour.c`, queues these ARP requests and the `neigh_proxy_process()` timer handler sends them back to `arp_rcv()`.

```c
1041 void pneigh_enqueue(struct neigh_table *tbl, struct
        neighParms *p, struct sk_buff *skb) {
1042     unsigned long now = jiffies;

1045     long sched_next = net_random() % p->proxy_delay;

1047     if (tbl->proxy_queue.qlen > p->proxy_qlen) {
1048         kfree_skb(skb);
1049         return;
1050     }

1051     skb->stamp.tv_sec = 0;
1052     skb->stamp.tv_usec = now + sched_next;
1054     spin_lock(&tbl->proxy_queue.lock);

1055     if (del_timer(&tbl->proxy_timer)) {
1056         long tval = tbl->proxy_timer.expires - now;
1057         if (tval < sched_next)
1058             sched_next = tval;
1059     }
```

The random proxy delay in jiffies is a random value in the range \(\{0,p->proxy_delay\}\) which is saved in the `sched_next` variable, which is the next scheduled delay of the proxy_timer.

If the proxy queue length is greater than the `proxy_qlen` parameter, skbuff is freed and returns back.

The `sk_buff` is time-stamped using the `tv_usec` field of struct timeval stamp field in it, with the sum of current clock ticks and the delay in `sched_next` field. The skbuff is dequeued only after this many clock ticks.

Delete the existing `proxy_timer` but if its scheduled expiration was before `sched_next` the previous value will be reused.
Release the next hop route destination present in the skbuff and increment the usage count of the device before queueing the skbuff in the proxy_queue.

```
1060    dst_release(skb->dst);
1061    skb->dst = NULL;
1062    dev_hold(skb->dev);
1063    __skb_queue_tail(&tbl->proxy_queue, skb);
```

The `mod_timer()` function is used to rearm the proxy_timer.

```
1064    mod_timer(&tbl->proxy_timer, now + sched_next);
1065    spin_unlock(&tbl->proxy_queue.lock);
1066 }
Proxy timer expiration

The `neigh_proxy_process()` function defined in net/core/neighbour.c is invoked on expiration of `proxy_timer`. (`arp_init` initialised `tbl->proxy_timer.function = neigh_proxy_process`)

```c
1007 static void neigh_proxy_process(unsigned long arg)
1008 {
1009     struct neigh_table *tbl = (struct neigh_table *)arg;
1010     long sched_next = 0;
1011     unsigned long now = jiffies;
1012     struct sk_buff *skb;
1013     spin_lock(&tbl->proxy_queue.lock);
1014 
1016     skb = tbl->proxy_queue.next;
1017     while (skb != (struct sk_buff*)&tbl->proxy_queue) {
1019         struct sk_buff *back = skb;
1020         long tdif = back->stamp.tv_usec - now;
1022         skb = skb->next;
1023         if (tdif <= 0) {
1024             struct net_device *dev = back->dev;
1025             __skb_unlink(back, &tbl->proxy_queue);
1026             if (tbl->proxy_redo && netif_running(dev))
1027                 tbl->proxy_redo(back);
1028             else
1029                 kfree_skb(back);
1031             dev_put(dev);
1032         } else if (!sched_next || tdif < sched_next)
1033             sched_next = tdif;
1034     }
1035     del_timer(&tbl->proxy_timer);
1036     if (sched_next)
1037         mod_timer(&tbl->proxy_timer, jiffies + sched_next);
1039 }
```
The standard proxy redo function

The `proxy_redo` function pointer in the `arp_tbl` structure refers to the `parp_redo()` function defined in `net/ipv4/arp.c`. It simply passes the `sk_buff` back to the `arp_rcv()` function.

```c
571 static void parp_redo(struct sk_buff *skb)  
572 {   
573         arp_rcv(skb, skb->dev, NULL);  
574 }
```