Chapitre 34

Envoi de trames

Comme nous venons de le voir dans le chapitre 33, les paquets IP sont transmis à la couche liaison de données pour être envoyés par un périphérique réseau grâce à la fonction `dev_queue_xmit(skb)`, où `skb` désigne un descripteur de tampon de socket dont la partie données comprend un paquet de la couche réseau.

Nous allons étudier dans ce chapitre comment les trames sont envoyées en prenant comme exemple la carte Ethernet 3Com 501.

Une fois que l'interface est activée, pour l'émission d'une trame le noyau se contente d'indiquer à la carte réseau qu'il y a des données à envoyer et leur emplacement en mémoire centrale. La carte s'occupe du reste, utilisant l'accès direct à la mémoire (DMA) pour récupérer les données. À la fin de l'émission, la carte lève une interruption pour prévenir le noyau que la trame a été envoyée avec succès ou que le délai s'est écoulé.

Les données consistent en un paquet de la couche réseau. La carte s'occupe d'encapsuler celui-ci pour obtenir une trame.

34.1 Transmission à la carte réseau : partie générale

La fonction `dev_queue_xmit()` traite la partie générale (c'est-à-dire non liée à un périphérique particulier) de l'envoi d'une trame. Elle appelle soit une fonction de traitement de la file d'attente des trames en partance, soit directement une fonction spécifique à la carte réseau utilisée. Autrement dit elle distingue deux cas :

- Si une méthode de gestion des file d'attente (`dev->enqueue != NULL`) est associée au périphérique, elle place le tampon de socket dans la file d'attente des trames en partance (grâce à la méthode `dev->qdisc->enqueue()`) puis elle fait appel à la méthode `qdisc_run()` pour déclencher la poursuite du traitement des paquets à envoyer.
- Sinon elle fait appel à l'émission immédiat du paquet (grâce à la fonction spécifique `dev->hard_start_xmit()`). Ce cas correspond aux périphériques réseau logiques comme le périphérique en boucle ou aux périphériques de tunnelisation.

34.1.1 Première étape : mise en file d'attente

34.1.1.1 Fonction de mise en file d'attente

Code Linux 2.6.10

La fonction `dev_queue_xmit()` est définie dans le fichier `linux/net/core/dev.c` :

```
1205 /**
1206  *      dev_queue_xmit - fait partir un tampon
1207  *      @skb : tampon a faire partir
1208  *
1209  *      Met en file d'attente un tampon pour qu'il soit transmis a un peripherique reseau.
1210  *      L'appelant doit avoir positionne le peripherique, la priorite et construit le
1211  *      tampon avant d'appeler cette fonction. La fonction peut etre appelee depuis une
        interruption.
1212  *
1213  *      Un code errno negatif est renvoye en cas d'echec. Un succes ne garantit pas
1214  *      que la trame sera transmise car elle peut etre detruite en cas de
1215  *      congestion ou de formation du trafic.
1216  */
1217
1218 int dev_queue_xmit(struct sk_buff *skb)
1219 {
1220         struct net_device *dev = skb->dev;
1221         struct Qdisc *q;
1222         int rc = -ENOMEM;
1223
1224         if (skb_shinfo(skb)->frag_list &&
1225             !(dev->features & NETIF_F_FRAGLIST) &&
1226             __skb_linearize(skb, GFP_ATOMIC))
1227                 goto out_kfree_skb;
1228
1229         /* Un skb fragmente est linearise si le peripherique ne supporte pas SG,
1230          * ou si au moins un des fragments est dans highmem et que le peripherique
1231          * ne supporte pas la DMA pour elle.
1232          */
1233         if (skb_shinfo(skb)->nr_frags &&
1234             (!(dev->features & NETIF_F_SG) || illegal_highdma(dev, skb)) &&
1235             __skb_linearize(skb, GFP_ATOMIC))
1236                 goto out_kfree_skb;
1237
1238         /* Si la somme de controle du paquet n'est pas calculee et que le peripherique
1239          * ne supporte pas le calcul pour ce protocole, l'effectuer ici.
1240          */
1241         if (skb->ip_summed == CHECKSUM_HW &&
```

```
1242             (!(dev->features & (NETIF_F_HW_CSUM | NETIF_F_NO_CSUM)) &&
1243              (!(dev->features & NETIF_F_IP_CSUM) ||
1244               skb->protocol != htons(ETH_P_IP))))
1245                 if (skb_checksum_help(skb, 0))
1246                         goto out_kfree_skb;
1247
1248         /* Annihiler les irqs logicielles pour divers verrous ci-dessous. Stopper
1249          * egalement la preemption pour RCU.
1250          */
1251         local_bh_disable();
1252
1253         /* Les mises a jour de qdisc sont serialisees par queue_lock.
1254          * La struct Qdisc qui est pointee par qdisc est maintenant une
1255          * structure rcu - on peut y acceder sans acquerir
1256          * de verrou (mais la structure peut ne pas etre a jour). La liberation du
1257          * qdisc sera retardee jusqu'a ce qu'il soit connu qu'il n'y a plus
1258          * aucune reference a lui.
1259          *
1260          * Si le qdisc a une fonction enqueue, nous aurons encore besoin de
1261          * detenir queue_lock avant de l'appeler, puisque queue_lock
1262          * serialise aussi l'acces a la file d'attente des peripheriques.
1263          */
1264
1265         q = rcu_dereference(dev->qdisc);
1266 #ifdef CONFIG_NET_CLS_ACT
1267         skb->tc_verd = SET_TC_AT(skb->tc_verd,AT_EGRESS);
1268 #endif
1269         if (q->enqueue) {
1270                 /* S'emparer de la file d'attente du peripherique */
1271                 spin_lock(&dev->queue_lock);
1272
1273                 rc = q->enqueue(skb, q);
1274
1275                 qdisc_run(dev);
1276
1277                 spin_unlock(&dev->queue_lock);
1278                 rc = rc == NET_XMIT_BYPASS ? NET_XMIT_SUCCESS : rc;
1279                 goto out;
1280         }
1281
1282         /* Le peripherique n'a pas de file d'attente. Cas frequent des peripheriques
1283            logiques : loopback, toutes sortes de tunnels...
1284
1285            C'est malheureusement pour cela que la protection xmit_lock est necessaire ici.
1286            (par exemple loopback et les tunnels IP sont propres, ignorant les
1287            compteurs de statistiques.)
1288            Il est possible cependant qu'ils reposent sur la protection
1289            effectuee par nous ici.
1290
1291            Verifier ceci et tirer le verrou. Ceci ne conduit pas a une impasse.
1292            Soit tirer le qdisc noqueue, ce qui est meme plus simple 8)
1293          */
1294         if (dev->flags & IFF_UP) {
1295                 int cpu = smp_processor_id(); /* ok because BHs are off */
1296
1297                 if (dev->xmit_lock_owner != cpu) {
1298
1299                         HARD_TX_LOCK(dev, cpu);
1300
1301                         if (!netif_queue_stopped(dev)) {
1302                                 if (netdev_nit)
1303                                         dev_queue_xmit_nit(skb, dev);
```

```
1304
1305                                rc = 0;
1306                                if (!dev->hard_start_xmit(skb, dev)) {
1307                                        HARD_TX_UNLOCK(dev);
1308                                        goto out;
1309                                }
1310                        }
1311                        HARD_TX_UNLOCK(dev);
1312                        if (net_ratelimit())
1313                                printk(KERN_CRIT "Virtual device %s asks to "
1314                                       "queue packet!\n", dev->name);
1315                } else {
1316                        /* Une recursion est detectee ! C'est possible,
1317                         * malheureusement */
1318                        if (net_ratelimit())
1319                                printk(KERN_CRIT "Dead loop on virtual device "
1320                                       "%s, fix it urgently!\n", dev->name);
1321                }
1322        }
1323
1324        rc = -ENETDOWN;
1325        local_bh_enable();
1326
1327 out_kfree_skb:
1328        kfree_skb(skb);
1329        return rc;
1330 out:
1331        local_bh_enable();
1332        return rc;
1333 }
```

Autrement dit :

- On déclare un descripteur de périphérique réseau, que l'on initialise avec celui associé au descripteur de tampon passé en argument.
- On déclare une stratégie de mise en file d'attente et un code de retour.
- On essaie de linéariser le tampon dans les cas indiqués en commentaire. Si on n'y parvient pas, on libère le descripteur de tampon et on renvoie l'opposé du code d'erreur `ENOMEM`.

Code Linux 2.6.10

La fonction `__skb_linearize()` est définie dans le fichier `linux/net/core/dev.c` :

```
1133 /* Garder la meme tete : remplacer les donnees */
1134 int __skb_linearize(struct sk_buff *skb, int gfp_mask)
1135 {
1136        unsigned int size;
1137        u8 *data;
1138        long offset;
1139        struct skb_shared_info *ninfo;
1140        int headerlen = skb->data - skb->head;
1141        int expand = (skb->tail + skb->data_len) - skb->end;
1142
1143        if (skb_shared(skb))
1144                BUG();
1145
1146        if (expand <= 0)
1147                expand = 0;
1148
1149        size = skb->end - skb->head + expand;
1150        size = SKB_DATA_ALIGN(size);
1151        data = kmalloc(size + sizeof(struct skb_shared_info), gfp_mask);
1152        if (!data)
1153                return -ENOMEM;
```

```
1154
1155         /* Copier la chose en entier */
1156         if (skb_copy_bits(skb, -headerlen, data, headerlen + skb->len))
1157                 BUG();
1158
1159         /* Initialiser shinfo */
1160         ninfo = (struct skb_shared_info*)(data + size);
1161         atomic_set(&ninfo->dataref, 1);
1162         ninfo->tso_size = skb_shinfo(skb)->tso_size;
1163         ninfo->tso_segs = skb_shinfo(skb)->tso_segs;
1164         ninfo->nr_frags = 0;
1165         ninfo->frag_list = NULL;
1166
1167         /* Le decalage entre les deux, en octets */
1168         offset = data - skb->head;
1169
1170         /* Liberer les vieilles donnees. */
1171         skb_release_data(skb);
1172
1173         skb->head = data;
1174         skb->end  = data + size;
1175
1176         /* Initialiser les nouveaux pointeurs */
1177         skb->h.raw   += offset;
1178         skb->nh.raw  += offset;
1179         skb->mac.raw += offset;
1180         skb->tail    += offset;
1181         skb->data    += offset;
1182
1183         /* Nous ne sommes plus un clone, meme si nous l'etions. */
1184         skb->cloned    = 0;
1185
1186         skb->tail     += skb->data_len;
1187         skb->data_len  = 0;
1188         return 0;
1189 }
```

La fonction en ligne `illegal_highdma()` est définie dans le fichier `linux/net/core/-dev.c` :

Code Linux 2.6.10

```
1108 #ifdef CONFIG_HIGHMEM
1109 /* En fait nous devons eliminer cette verification des que nous savons que :
1110  * 1. IOMMU est present et permet d'adresser toute la memoire.
1111  * 2. Aucune memoire haute n'existe sur cette machine.
1112  */
1113
1114 static inline int illegal_highdma(struct net_device *dev, struct sk_buff *skb)
1115 {
1116         int i;
1117
1118         if (dev->features & NETIF_F_HIGHDMA)
1119                 return 0;
1120
1121         for (i = 0; i < skb_shinfo(skb)->nr_frags; i++)
1122                 if (skb_shinfo(skb)->frags[i].page >= highmem_start_page)
1123                         return 1;
1124
1125         return 0;
1126 }
1127 #else
1128 #define illegal_highdma(dev, skb)       (0)
1129 #endif
```

– Si la somme de contrôle du paquet n'a pas été calculée et que le périphérique ne supporte pas le calcul de celle-ci pour ce protocole, on la calcule. Si on n'y parvient pas, on libère le descripteur de tampon et on renvoie l'opposé du code d'erreur `ENOMEM`.

Code Linux 2.6.10

La fonction `skb_checksum_help()` est définie dans le fichier `linux/net/core/dev.c`:

```
1072 /*
1073  * Invalide la somme de controle du materiel lorsque le paquet doit etre estropie, et
1074  * calcule la somme complete en chemin.
1075  */
1076 int skb_checksum_help(struct sk_buff *skb, int inward)
1077 {
1078         unsigned int csum;
1079         int ret = 0, offset = skb->h.raw - skb->data;
1080
1081         if (inward) {
1082                 skb->ip_summed = CHECKSUM_NONE;
1083                 goto out;
1084         }
1085
1086         if (skb_cloned(skb)) {
1087                 ret = pskb_expand_head(skb, 0, 0, GFP_ATOMIC);
1088                 if (ret)
1089                         goto out;
1090         }
1091
1092         if (offset > (int)skb->len)
1093                 BUG();
1094         csum = skb_checksum(skb, offset, skb->len-offset, 0);
1095
1096         offset = skb->tail - skb->h.raw;
1097         if (offset <= 0)
1098                 BUG();
1099         if (skb->csum + 2 > offset)
1100                 BUG();
1101
1102         *(u16*)(skb->h.raw + skb->csum) = csum_fold(csum);
1103         skb->ip_summed = CHECKSUM_NONE;
1104 out:
1105         return ret;
1106 }
```

– On annihile les interruptions logicielles.

– Si la stratégie de mise en file d'attente associée au descripteur de périphérique possède une fonction d'empilement : on verrouille la file d'attente du périphérique, on place le descripteur de tampon dans la file d'attente grâce à la fonction `enqueue()` de cette stratégie, on fait appel à la fonction `qdisc_run()` étudiée ci-après, on déverrouille la file d'attente et on renvoie l'une des constantes symboliques `NET_XMIT_BYPASS` ou `NET_XMIT_SUCCESS`.

Code Linux 2.6.10

Celles-ci sont définies dans le fichier `linux/include/linux/netdevice.h`:

```
54 #define NET_XMIT_SUCCESS        0
55 #define NET_XMIT_DROP           1       /* skb ecarte                          */
56 #define NET_XMIT_CN             2       /* notification de congestion          */
57 #define NET_XMIT_POLICED        3       /* skb est abattu par la police        */
58 #define NET_XMIT_BYPASS         4       /* le paquet ne quitte pas via dequeue ;
59                                            (usage TC seulement - dev_queue_xmit
60                                            renvoie ceci comme NET_XMIT_SUCCESS) */
```

– S'il n'y a pas de fonction d'empilement mais que le périphérique est actif, on lui indique qu'on vient de placer de nouvelles données en faisant appel tout de suite à la fonction d'émission `dev->hard_start_xmit(skb, dev)` spécifique à la carte.

34.1.1.2 Empilement

La fonction précédente fait référence à la méthode `enqueue()` de la stratégie de mise en file d'attente. Nous avons vu au chapitre 18 que `enqueue()` peut prendre l'une des deux valeurs `noop_enqueue()` ou `pfifo_fast_enqueue()`.

Cas de la stratégie qui ne fait rien

La fonction `noop_enqueue()` est définie dans le fichier `linux/net/sched/sch_generic.c`: Code Linux 2.6.10

```
240 /* Ordonnanceur "NOOP" : le meilleur ordonnanceur, recommande pour toutes les interfaces
241    et en toute circonstance. Il est difficile d'inventer une chose plus rapide
242    ou moins chere.
243  */
244
245 static int
246 noop_enqueue(struct sk_buff *skb, struct Qdisc * qdisc)
247 {
248         kfree_skb(skb);
249         return NET_XMIT_CN;
250 }
```

Elle libère le descripteur de tampon passé en argument et indique que la file d'attente est congestionnée.

Cas de la stratégie rapide

Plus sérieusement, la fonction `pfifo_fast_enqueue()` est définie dans le fichier `linux/net/sched/sch_generic.c`: Code Linux 2.6.10

```
306 static const u8 prio2band[TC_PRIO_MAX+1] =
307         { 1, 2, 2, 2, 1, 2, 0, 0 , 1, 1, 1, 1, 1, 1, 1, 1 };
308
309 /* Une file d'attente FIFO a 3 bandes : style ancien mais devrait etre un petit peu
310    plus rapide que la combinaison prio+fifo generique.
311  */
312
313 static int
314 pfifo_fast_enqueue(struct sk_buff *skb, struct Qdisc* qdisc)
315 {
316         struct sk_buff_head *list = qdisc_priv(qdisc);
317
318         list += prio2band[skb->priority&TC_PRIO_MAX];
319
320         if (list->qlen < qdisc->dev->tx_queue_len) {
321                 __skb_queue_tail(list, skb);
322                 qdisc->q.qlen++;
323                 qdisc->bstats.bytes += skb->len;
324                 qdisc->bstats.packets++;
325                 return 0;
326         }
327         qdisc->qstats.drops++;
328         kfree_skb(skb);
329         return NET_XMIT_DROP;
330 }
```

La constante symbolique `TC_PRIO_MAX` est définie dans le fichier `linux/include/linux/pkt_sched.h`: Code Linux 2.6.10

```
4  /* Bandes de priorite logique ne dependant pas d'un ordonnanceur de paquets particulier.
5     Chaque ordonnanceur les appliquera aux classes de trafic reelles s'il n'a pas
6     de mecanisme plus precis pour classifier les paquets.
7
8     Ces nombres n'ont pas de signification speciale, bien que leur coincidence
```

```
9     avec les valeurs IPv6 obsoletes ne soit pas accidentelle :-). Les nouvelles esquisses
10    de IPv6 ont prefere l'anarchie totale inspiree par differents groupes.
11
12    Note : TC_PRIO_BESTEFFORT ne signifie pas qu'il s'agisse de la classe la plus malheureuse
13    actuellement, en regle generale elle sera manipulee avec plus de soin qu'un
14    entonnoir ou de meme grosseur.
15  */
16
17 #define TC_PRIO_BESTEFFORT              0
18 #define TC_PRIO_FILLER                  1
19 #define TC_PRIO_BULK                    2
20 #define TC_PRIO_INTERACTIVE_BULK        4
21 #define TC_PRIO_INTERACTIVE             6
22 #define TC_PRIO_CONTROL                 7
23
24 #define TC_PRIO_MAX                     15
```

34.1.2 Deuxième étape : récupération des paquets

34.1.2.1 Fonction principale

La fonction `qdisc_run()` comporte peu de fonctionnalités. Elle appelle seulement la fonction `qdisc_restart()` tant que celle-ci retourne une valeur de renvoi supérieure ou égale à zéro, c'est-à-dire jusqu'à ce qu'il n'y ait plus de paquets dans la file d'attente, ou que le périphérique réseau n'accepte plus de paquet (ce que l'on sait grâce à la fonction `netif_queue_stopped(dev)`).

Code Linux 2.6.10

La fonction `qdisc_run()` est définie dans le fichier `linux/include/net/pkt_sched.h`:

```
231 static inline void qdisc_run(struct net_device *dev)
232 {
233         while (!netif_queue_stopped(dev) && qdisc_restart(dev) < 0)
234                 /* RIEN */;
235 }
```

Code Linux 2.6.10

La fonction `netif_queue_stopped()` est définie dans le fichier `linux/include/linux/net-device.h`:

```
630 static inline int netif_queue_stopped(const struct net_device *dev)
631 {
632         return test_bit(__LINK_STATE_XOFF, &dev->state);
633 }
```

34.1.2.2 Récupération et émission d'un paquet

La fonction `qdisc_restart()` est chargée de récupérer le paquet suivant dans la file d'attente en émission du périphérique réseau et de l'envoyer. La valeur de retour :

- est nulle lorsque la file d'attente est vide.
- est strictement positive lorsque la file d'attente n'est pas vide mais que, conformément à la stratégie de mise en file d'attente (`dev->qdisc`), aucun paquet ne peut être envoyé, par exemple parce qu'il n'a pas encore atteint son moment de transmission désiré (mise en forme du trafic).
- est strictement négative lorsque la file d'attente n'est pas vide mais que le périphérique réseau ne peut plus en ce moment accepter de paquet parce que tous les emplacements du tampon sont occupés.

Code Linux 2.6.10

La fonction `qdisc_restart()` est définie dans le fichier `linux/net/sched/sch_generic.c`:

```
82  /* Donne un coup de pied au peripherique.
```

```
83      Noter que cette procedure peut etre appelee par un minuteur chien de garde, et
84      donc que nous ne verifions pas le drapeau dev->tbusy ici.
85
86      Renvoie :  0  - la file d'attente est vide.
87                >0  - la file d'attente n'est pas vide mais ralentie.
88                <0  - la file d'attente n'est pas vide. Le peripherique est ralenti
                        si dev->tbusy != 0.
89
90      NOTE : Appelee sous dev->queue_lock avec BH localement inhibee.
91   */
92
93  int qdisc_restart(struct net_device *dev)
94  {
95          struct Qdisc *q = dev->qdisc;
96          struct sk_buff *skb;
97
98          /* Retire le paquet de la file d'attente */
99          if ((skb = q->dequeue(q)) != NULL) {
100                 unsigned nolock = (dev->features & NETIF_F_LLTX);
101                 /*
102                  * Lorsque le pilote a LLTX positionne, il fait son propre verrouillage
103                  * dans start_xmit. Pas besoin d'ajouter encore des verrouillages
104                  * supplementaires. Ces verifications sont de valeur puisque
105                  * des verrous meme non congestionnes peuvent etre tres onereux.
106                  * Le pilote peut faire des trylocks comme ici aussi, en cas
107                  * de congestion de verrou, il renverra -1 et le paquet
108                  * sera replace dans la file d'attente.
109                  */
110                 if (!nolock) {
111                         if (!spin_trylock(&dev->xmit_lock)) {
112                         collision:
113                                 /* Ainsi quelqu'un a accapare le pilote. */
114
115                                 /* Cela peut etre une erreur de configuration transitoire,
116                                    lorsque hard_start_xmit() est recursif. Nous le detectons
117                                    en verifiant le proprietaire en emission et nous ecartons
118                                    le paquet lorsque le deadloop est detecte.
119                                 */
120                                 if (dev->xmit_lock_owner == smp_processor_id()) {
121                                         kfree_skb(skb);
122                                         if (net_ratelimit())
123                                                 printk(KERN_DEBUG "Dead loop on netdevice %s,
                                                                fix it urgently!\n", dev->name);
124                                         return -1;
125                                 }
126                                 __get_cpu_var(netdev_rx_stat).cpu_collision++;
127                                 goto requeue;
128                         }
129                         /* Rappelons que nous avons accapare le pilote. */
130                         dev->xmit_lock_owner = smp_processor_id();
131                 }
132
133                 {
134                         /* Et relachons la file d'attente */
135                         spin_unlock(&dev->queue_lock);
136
137                         if (!netif_queue_stopped(dev)) {
138                                 int ret;
139                                 if (netdev_nit)
140                                         dev_queue_xmit_nit(skb, dev);
141
142                                 ret = dev->hard_start_xmit(skb, dev);
```

```
143                                 if (ret == NETDEV_TX_OK) {
144                                         if (!nolock) {
145                                                 dev->xmit_lock_owner = -1;
146                                                 spin_unlock(&dev->xmit_lock);
147                                         }
148                                         spin_lock(&dev->queue_lock);
149                                         return -1;
150                                 }
151                                 if (ret == NETDEV_TX_LOCKED && nolock) {
152                                         spin_lock(&dev->queue_lock);
153                                         goto collision;
154                                 }
155                         }
156
157                         /* NETDEV_TX_BUSY - nous avons besoin de le replacer dans la file
                               d'attente */
158                         /* Liberons le pilote */
159                         if (!nolock) {
160                                 dev->xmit_lock_owner = -1;
161                                 spin_unlock(&dev->xmit_lock);
162                         }
163                         spin_lock(&dev->queue_lock);
164                         q = dev->qdisc;
165                 }
166
167                 /* Le peripherique nous donne un coup de pied :(
168                    Ceci est possible dans trois cas :
169
170                    0. le peripherique est verrouille
171                    1. fastroute est active
172                    2. le peripherique ne peut pas determiner un etat occupe
173                       avant le debut de la transmission (par exemple par telephone)
174                    3. le peripherique est bogue (ppp)
175                  */
176
177 requeue:
178                 q->ops->requeue(skb, q);
179                 netif_schedule(dev);
180                 return 1;
181         }
182         return q->q.qlen;
183 }
```

Remarquons que la référence à `dev->tbusy` dans le commentaire n'a plus lieu d'etre puisque ce champ n'existe plus depuis la version 2.6.0 du noyau.
Autrement dit :

- On déclare une stratégie de mise en file d'attente, que l'on initialise avec celle associée au descripteur de périphérique réseau passé en argument.
- On déclare un descripteur de tampon de socket.
- On essaie de récupérer un descripteur de tampon dans la file d'attente.
- Si on y parvient et si le pilote n'a pas déjà verrouillé, on essaie de positionner le verrou rotatif en émission du descripteur de périphérique. Si on n'y parvient pas, c'est qu'un microprocesseur diffuse un autre paquet *via* ce périphérique de réseau :
  - S'il s'agit du même microprocesseur, on rejette le paquet : on libère le descripteur de tampon, on affiche un message noyau et on renvoie -1.
  - S'il s'agit d'un autre microprocesseur, on incrémente le nombre de collisions liées à des problèmes de microprocesseurs, le tampon est replacé dans la file d'attente, l'envoi

du paquet est différé, ce qui est effectué en faisant appel à la fonction `netif_schedule(dev)`, étudiée ci-dessous et on renvoie 1.

La variable `netdev_rx_stat` est déclarée dans le fichier `linux/net/core/dev.c`: Code Linux 2.6.10

```
1351 DEFINE_PER_CPU(struct netif_rx_stats, netdev_rx_stat) = { 0, };
```

Le type `netif_rx_stats` est déclaré dans le fichier `linux/include/linux/netdevice.h`: Code Linux 2.6.10

```
162 struct netif_rx_stats
163 {
164         unsigned total;
165         unsigned dropped;
166         unsigned time_squeeze;
167         unsigned throttled;
168         unsigned fastroute_hit;
169         unsigned fastroute_success;
170         unsigned fastroute_defer;
171         unsigned fastroute_deferred_out;
172         unsigned fastroute_latency_reduction;
173         unsigned cpu_collision;
174 };
```

Si on y parvient, on spécifie au descripteur de périphérique le microprocesseur qui a verrouilé la file d'attente.

- On déverrouille la file d'attente du descripteur de périphérique.
- Si le périphérique réseau accepte un paquet :
  - Si la variable `netdev_nit` est positionnée, on fait appel à la fonction `dev_queue_xmit_nit(skb, dev)`, étudiée ci-après, d'envoi à toutes les cartes.

    La variable `netdev_nit` est déclarée dans le fichier `linux/net/core/dev.c`: Code Linux 2.6.10

    ```
    236 /*
    237  *      Pour l'efficacite
    238  */
    239
    240 int netdev_nit;
    ```

  - On fait appel à la fonction spécifique à la carte réseau `dev->hard_start_xmit(skb, dev)` pour envoyer le paquet.

    Les constantes symboliques `NETDEV_TX_OK` et `NETDEV_TX_LOCKED` sont définies dans le fichier `linux/include/linux/netdevice.h`: Code Linux 2.6.10

    ```
    76 /* Codes de retour pour l'emission de la part d'un pilote */
    77 #define NETDEV_TX_OK 0          /* Le pilote a pris soin du paquet */
    78 #define NETDEV_TX_BUSY 1        /* Le chamin en tx du pilote etait occupe */
    79 #define NETDEV_TX_LOCKED -1     /* Le verrou en tx du pilote etait deja tire */
    ```

  - Si le pilote a transmis le paquet au périphérique, on indique éventuellement qu'aucun microprocesseur n'a accaparé le périphérique et on déverrouille en émission, on verrouille la file d'attente et on renvoie -1.
  - Si le verrou en émission du pilote était déjà tiré, on verrouille la file d'attente et on continue comme dans le cas d'une collision avec un autre microprocesseur.
- Si le périphérique réseau n'accepte pas de nouvelle trame pour le moment, on indique éventuellement qu'aucun microprocesseur n'a accaparé le périphérique et on déverrouille en émission, on verrouille la file d'attente et on initialise la stratégie de mise en file d'attente avec celle du descripteur de périphérique passé en argument.

– Si on ne parvient pas à récupérer de paquet dans la file d'attente, on renvoie la longueur de la file d'attente.

En conclusion, dans le cas d'un envoi normal, il est fait appel à la fonction spécifique à la carte réseau pour envoyer le paquet et on se prépare à envoyer le paquet suivant.

34.1.2.3 Fonction de dépilement

La fonction précédente fait référence à la méthode `dequeue()` de la stratégie de file d'attente. Nous avons vu au chapitre 18 que `enqueue()` peut prendre l'une des deux valeurs `noop_dequeue()` ou `pfifo_fast_dequeue()`.

Stratégie qui ne fait rien

Code Linux 2.6.10

La fonction `noop_dequeue()` est définie dans le fichier `linux/net/sched/sch_generic.c`:

```
252 static struct sk_buff *
253 noop_dequeue(struct Qdisc * qdisc)
254 {
255         return NULL;
256 }
```

Stratégie rapide

Code Linux 2.6.10

La fonction `pfifo_fast_dequeue()` est définie dans le fichier `linux/net/sched/sch_generic.c`:

```
332 static struct sk_buff *
333 pfifo_fast_dequeue(struct Qdisc* qdisc)
334 {
335         int prio;
336         struct sk_buff_head *list = qdisc_priv(qdisc);
337         struct sk_buff *skb;
338
339         for (prio = 0; prio < 3; prio++, list++) {
340                 skb = __skb_dequeue(list);
341                 if (skb) {
342                         qdisc->q.qlen--;
343                         return skb;
344                 }
345         }
346         return NULL;
347 }
```

34.1.2.4 Fonction de réempilement

La fonction précédente fait également référence à la méthode `requeue()` de la stratégie de file d'attente. Nous avons vu au chapitre 18 que `requeue()` peut prendre l'une des deux valeurs `noop_requeue()` ou `pfifo_fast_requeue()`.

Stratégie qui ne fait rien

Code Linux 2.6.10

La fonction `noop_requeue()` est définie dans le fichier `linux/net/sched/sch_generic.c`:

```
258 static int
259 noop_requeue(struct sk_buff *skb, struct Qdisc* qdisc)
260 {
261         if (net_ratelimit())
262                 printk(KERN_DEBUG "%s deferred output. It is buggy.\n", skb->dev->name);
263         kfree_skb(skb);
264         return NET_XMIT_CN;
265 }
```

Stratégie rapide

La fonction `pfifo_fast_requeue()` est définie dans le fichier `linux/net/sched/sch_generic.c` :

Code Linux 2.6.10

```
349 static int
350 pfifo_fast_requeue(struct sk_buff *skb, struct Qdisc* qdisc)
351 {
352         struct sk_buff_head *list = qdisc_priv(qdisc);
353
354         list += prio2band[skb->priority&TC_PRIO_MAX];
355
356         __skb_queue_head(list, skb);
357         qdisc->q.qlen++;
358         qdisc->qstats.requeues++;
359         return 0;
360 }
```

34.1.3 Troisième étape : envoi

Outre l'envoi direct au périphérique, nous avons vu que l'étape précédente fait appel à l'envoi différé et à l'envoi à toutes les cartes.

34.1.3.1 Envoi différé

La fonction `netif_schedule()` est définie dans le fichier `linux/include/linux/netdevice.h` :

Code Linux 2.6.10

```
585 static inline void __netif_schedule(struct net_device *dev)
586 {
587         if (!test_and_set_bit(__LINK_STATE_SCHED, &dev->state)) {
588                 unsigned long flags;
589                 struct softnet_data *sd;
590
591                 local_irq_save(flags);
592                 sd = &__get_cpu_var(softnet_data);
593                 dev->next_sched = sd->output_queue;
594                 sd->output_queue = dev;
595                 raise_softirq_irqoff(NET_TX_SOFTIRQ);
596                 local_irq_restore(flags);
597         }
598 }
599
600 static inline void netif_schedule(struct net_device *dev)
601 {
602         if (!test_bit(__LINK_STATE_XOFF, &dev->state))
603                 __netif_schedule(dev);
604 }
```

Autrement dit si le périphérique n'est pas désactivé, on fait appel à la fonction `__netif_schedule()` définie juste avant dans le même fichier. Celle-ci :

- ne fait rien si le périphérique est déjà dans l'état “ordonnancement” ;
- sinon on déclare un vecteur de drapeaux et une file d'attente en émission de microprocesseur ;
- on inhibe les interruptions ;
- on instantie la file d'attente avec celle du microprocesseur en cours ;
- on ajoute le descripteur de périphérique passé en argument à cette file d'attente ;
- on lève `NET_TX_SOFTIRQ` ;
- et on restaure les interruptions matérielles.

34.1.3.2 Envoi à toutes les cartes

Code Linux 2.6.10

La fonction `dev_queue_xmit_nit()` est définie dans le fichier `linux/net/core/dev.c`:

```
1027 /*
1028  *      Routine de support. Envoie les trames de sortie a toutes les cartes
1029  *      reseau actuellement actives.
1030  */
1031
1032 void dev_queue_xmit_nit(struct sk_buff *skb, struct net_device *dev)
1033 {
1034         struct packet_type *ptype;
1035         net_timestamp(&skb->stamp);
1036
1037         rcu_read_lock();
1038         list_for_each_entry_rcu(ptype, &ptype_all, list) {
1039                 /* Ne jamais renvoyer les paquets a la socket
1040                  * dont ils proviennent - MvS (miquels@drinkel.ow.org)
1041                  */
1042                 if ((ptype->dev == dev || !ptype->dev) &&
1043                     (ptype->af_packet_priv == NULL ||
1044                      (struct sock *)ptype->af_packet_priv != skb->sk)) {
1045                        struct sk_buff *skb2= skb_clone(skb, GFP_ATOMIC);
1046                        if (!skb2)
1047                                break;
1048
1049                        /* skb->nh doit etre correctement positionne
1050                           par l'emetteur, ce second enonce est donc
1051                           juste une protection contre les protocoles bogues.
1052                         */
1053                        skb2->mac.raw = skb2->data;
1054
1055                        if (skb2->nh.raw < skb2->data ||
1056                            skb2->nh.raw > skb2->tail) {
1057                                if (net_ratelimit())
1058                                        printk(KERN_CRIT "protocol %04x is "
1059                                               "buggy, dev %s\n",
1060                                               skb2->protocol, dev->name);
1061                                skb2->nh.raw = skb2->data;
1062                        }
1063
1064                        skb2->h.raw = skb2->nh.raw;
1065                        skb2->pkt_type = PACKET_OUTGOING;
1066                        ptype->func(skb2, skb->dev, ptype);
1067                }
1068         }
1069         rcu_read_unlock();
1070 }
```

Autrement dit :

- On déclare un type de paquets.
- On renseigne le champ estampille du descripteur de tampon passé en argument avec l'heure en cours.

Code Linux 2.6.10

La fonction `net_timestamp()` est définie dans le fichier `linux/net/core/dev.c`:

```
1017 static inline void net_timestamp(struct timeval *stamp)
1018 {
1019         if (atomic_read(&netstamp_needed))
1020                 do_gettimeofday(stamp);
1021         else {
1022                 stamp->tv_sec = 0;
```

```
1023                    stamp->tv_usec = 0;
1024            }
1025 }
```

- On verrouille la mise à jour en écriture.
- On parcourt la liste de tous les types de paquets à la recherche des types dont le champ descripteur de périphérique est égal à celui passé en argument ou non spécifié et dont le champ type de paquet n'est pas spécifié ou correspond à celui du descripteur de tampon passé en argument. Pour chacun de ces types de paquet :
  - On essaie de cloner le descripteur de tampon passé en argument. Si on n'y parvient pas, on passe au type suivant.
  - On renseigne certains champs de ce descripteur de tampon : le champ en-tête matériel (brut) avec le début des données ; si le champ en-tête réseau brut n'est pas bien positionné (ce qui ne devrait pas être le cas comme l'indique le commentaire), on renseigne également ce champ avec le début des données ; le champ en-tête transport (brut) avec l'en-tête réseau ; le champ type de paquet avec "paquet sortant".
  - On fait appel à la fonction spécifique de ce type de paquet pour l'envoi proprement dit.
- On déverrouille la mise à jour en écriture.

34.2 Transmission d'un paquet à la carte : partie spécifique à la carte

34.2.1 Étude générale

Les fonctions précédentes font appel à la fonction `dev->hard_start_xmit(skb, dev)` spécifique à la carte réseau. Celle-ci est chargée de copier la zone des données du tampon de socket dans un emplacement interne du périphérique et de commander le début de la transmission. Si la copie réussit, on part également du principe que l'émission réussira. Dans ce cas, `hard_start_xmit()` doit revenir à l'état antérieur avec la valeur de renvoi 0. Dans le cas contraire, un 1 sera envoyé pour que le noyau sache que l'émission du paquet n'a pas pu aboutir.

Lors de la remise des paquets, on peut distinguer deux techniques de copie entre système d'exploitation et adaptateur réseau :

- Les cartes les plus anciennes, comme la carte 3c501 de 3Com, comportent une mémoire tampon interne en prévision de l'émission de paquets sur l'adaptateur. Ainsi, le noyau ne peut-il remettre qu'un seul paquet à l'adaptateur. Il copie alors le paquet dans un tampon libre à l'identique de l'adaptateur et le noyau peut supprimer le tampon de socket correspondant.
- Les adaptateurs réseau plus récents suivent une autre voie. Le pilote gère un tampon circulaire de 16 à 64 pointeurs de tampon de socket. Lorsqu'il s'agit d'émettre un paquet, le tampon de socket correspondant est placé dans cet anneau et un pointeur sur la zone des données est transmis à l'adaptateur de réseau. Le tampon de socket reste ensuite dans le tampon circulaire jusqu'à ce que l'adaptateur ait signalé son émission par une interruption. Le tampon de socket est alors retiré et libéré du tampon circulaire.

  En cas de succès, le tampon de socket n'est plus nécessaire et peut être libéré. En cas d'échec, il ne faut pas toucher au tampon de socket parce que le noyau tentera probablement de l'émettre à nouveau.

34.2.2 Cas de la carte 3Com 501 : Mise en file d'attente

Nous avons vu au chapitre 17 que la fonction `hard_start_xmit()` spécifique à la carte Ethernet 3Com 501 s'appelle `el_start_xmit()`. Celle-ci est définie dans le fichier `linux/drivers/net/3c501.c`:

Code Linux 2.6.10

```
390 /**
391  * el_start_xmit :
392  * @skb : Le paquet qui est mis en file d'attente pour etre envoye
393  * @dev : La carte 3c501 dans laquelle nous voulons le deposer
394  *
395  * Essaie d'envoyer un paquet a une carte 3c501. Il y a certaines entourloupettes
396  * interessantes ici puisque la 3c501 est extrement vieille et donc une piece de
397  * technologie stupide.
398  *
399  * Si nous traitons une interruption sur l'autre CPU, nous ne pouvons pas charger un paquet
400  * alors que nous pouvons encore essayer de recuperer le dernier tampon de paquet en RX.
401  *
402  * Lorsqu'une emission a depasse le delai, nous vidons la carte du mode de controle
403  * et nous redemarrons. Il arrive souvent que ce ne soit pas pire.
404  *
405  * Nous evitons de detenir les verrous tournants lorsque le paquet est en train de
406  * passer dans la carte. Le peripherique est tres lent, et son mode DMA est encore plus
407  * lent. Si nous detenions le verrou lors du transfert des 1500 octets dans le controleur,
408  * nous ecarterions beaucoup trop de caracteres du port serie. Ceci exige que nous
409  * fassions un verrouillage supplementaire, mais nous n'avons pas reellement le choix.
410  */
411
412 static int el_start_xmit(struct sk_buff *skb, struct net_device *dev)
413 {
414         struct net_local *lp = netdev_priv(dev);
415         int ioaddr = dev->base_addr;
416         unsigned long flags;
417
418         /*
419          *      Eviter que ne surviennent des interruptions entre nos petites emissions et le
420          *      mode ou le peripherique suppose que la transmission est un indicateur fidele
421          *      de l'etat de la carte
422          */
423
424         spin_lock_irqsave(&lp->lock, flags);
425
426         /*
427          *      Eviter les conflits de retransmission fondes sur le delai.
428          */
429
430         netif_stop_queue(dev);
431
432         do
433         {
434                 int len = skb->len;
435                 int pad = 0;
436                 int gp_start;
437                 unsigned char *buf = skb->data;
438
439                 if (len < ETH_ZLEN)
440                         pad = ETH_ZLEN - len;
441
442                 gp_start = 0x800 - ( len + pad );
443
444                 lp->tx_pkt_start = gp_start;
445                 lp->collisions = 0;
446
```

```
447                 lp->stats.tx_bytes += skb->len;
448
449                 /*
450                  *      Le mode commande avec le statut efface devrait [en theorie]
451                  *      signifier qu'il n'y a plus d'interruptions en attente sur la carte.
452                  */
453
454                 outb_p(AX_SYS, AX_CMD);
455                 inb_p(RX_STATUS);
456                 inb_p(TX_STATUS);
457
458                 lp->loading = 1;
459                 lp->txing = 1;
460
461                 /*
462                  *      Retourne aux interruptions cependant que nous passons une apres-midi
463                  *      charmante en train de charger des octets dans la carte
464                  */
465
466                 spin_unlock_irqrestore(&lp->lock, flags);
467
468                 outw(0x00, RX_BUF_CLR);         /* Met la zone du paquet en rx a 0. */
469                 outw(gp_start, GP_LOW);         /* but - le paquet sera charge au
                                                       debut du tampon */
470                 outsb(DATAPORT,buf,len);        /* charge le tampon (chose usuelle, chaque
                                                       octets incremente le pointeur) */
471                 if (pad) {
472                         while(pad--)            /* Fin du tampon rempli avec des zeros */
473                                 outb(0, DATAPORT);
474                 }
475                 outw(gp_start, GP_LOW);         /* la carte reutilise le meme registre */
476
477                 if(lp->loading != 2)
478                 {
479                         outb(AX_XMIT, AX_CMD);          /* feu... envoyer l'emission.  */
480                         lp->loading=0;
481                         dev->trans_start = jiffies;
482                         if (el_debug > 2)
483                                 printk(KERN_DEBUG " queued xmit.\n");
484                         dev_kfree_skb (skb);
485                         return 0;
486                 }
487                 /* Une reception recoit notre chargement en depit de nos meilleurs efforts */
488                 if(el_debug>2)
489                         printk(KERN_DEBUG "%s: burped during tx load.\n", dev->name);
490                 spin_lock_irqsave(&lp->lock, flags);
491         }
492         while(1);
493
494 }
```

Autrement dit :

- On déclare une structure de réseau locale, que l'on initialise avec la partie privée du descripteur de périphérique réseau passé en argument.
- On déclare un port d'entrée-sortie, que l'on initialise avec celui fourni par le descripteur de périphérique réseau passé en argument.
- On déclare un vecteur de drapeaux.
- On sauvegarde les registres IRQ et on inhibe les interruptions.
- On évite les conflits de retransmission en arrêtant la file d'attente du périphérique.

Code Linux 2.6.10

La fonction `netif_stop_queue()` est définie dans le fichier `linux/include/linux/netdevice.h`:

```
621 static inline void netif_stop_queue(struct net_device *dev)
622 {
623 #ifdef CONFIG_NETPOLL_TRAP
624         if (netpoll_trap())
625                 return;
626 #endif
627         set_bit(__LINK_STATE_XOFF, &dev->state);
628 }
```

- On entre dans une boucle (*a priori* infinie) de scrutation:
  - On déclare une taille, que l'on initialise avec la taille du tampon de socket passé en argument.
  - On déclare une valeur de rembourrage, que l'on initialise à zéro.
  - On déclare une valeur entière `gp_start`, qui est la valeur du pointeur `GP` de la carte.
  - On déclare une adresse de chaîne de caractères, que l'on initialise avec celle des données du tampon de socket passé en argument.
  - Si la taille est strictement inférieure à celle d'une trame Ethernet minimum, on calcule le nombre de caractères de remplissage.
  - On calcule la valeur de `GP` à transmettre à la carte.
  - On initialise la structure locale de réseau: valeur de `GP`, nombre de collisions nul, le nombre d'octets qui restent à transmettre.
  - On réinitialise le mode de commande de la carte.
  - On indique à la structure de réseau locale que l'on est en train de charger des données en vue d'une émission.
  - On remet les interruptions en service.
  - On réinitialise le pointeur de la zone de réception de la carte réseau en la mettant à zéro.
  - On indique à la carte l'emplacement de mémoire vive depuis lequel charger le paquet à transmettre.
  - On transfère le paquet de la mémoire vive vers la carte réseau, en ajoutant si besoin est des caractères de remplissage.
  - On positionne le pointeur `GP` de la carte.
  - Si le niveau de chargement est différent de 2:
    - On envoie à la carte l'ordre d'émettre.
    - On passe au niveau de chargement 0.
    - On indique au descripteur de périphérique l'heure de la dernière transmission.
    - Si le niveau de débogage est strictement supérieur à 2, on affiche un message noyau.
    - On libère le descripteur de tampon.
    - On renvoie 0 (et donc on sort de la boucle infinie).
  - Si on n'est pas parvenu à émettre: si le niveau de débogage est strictement supérieur à 2, on affiche un message noyau, on verrouille l'interruption et on recommence.

34.3 Réaction à l'envoi

Une fois que le paquet a été transmis à la file d'attente en émission de la carte, on attend une réaction à cet envoi : soit une interruption spécifiant que la trame a été envoyée avec succès, soit la fin du délai.

34.3.1 Réponse du gestionnaire d'interruption

Le gestionnaire d'interruption dépend de la carte réseau. Nous allons nous intéresser, comme d'habitude, à celui de la carte 3Com 501. Nous avons commencer à étudier cette routine dans le chapitre consacré à la réception. Reprenons notre commentaire de la routine d'interruption dans le cas de l'émission :

Code Linux 2.6.10

```
519 static irqreturn_t el_interrupt(int irq, void *dev_id, struct pt_regs *regs)
520 {
[...]
548         if (lp->txing)
549         {
550
551                 /*
552                  *      La carte est en mode emission. Peut etre en train de charger.
553                  *      Si nous sommes en train de charger, nous devrions avoir obtenu ceci.
554                  */
555
556                 int txsr = inb(TX_STATUS);
557
558                 if(lp->loading==1)
559                 {
560                         if(el_debug > 2)
561                         {
562                                 printk(KERN_DEBUG "%s: Interrupt while loading [", dev->name);
563                                 printk(KERN_DEBUG " txsr=%02x gp=%04x rp=%04x]\n", txsr,
                                        inw(GP_LOW),inw(RX_LOW));
564                         }
565                         lp->loading=2;          /* Force a charger a nouveau */
566                         spin_unlock(&lp->lock);
567                         goto out;
568                 }
569
570                 if (el_debug > 6)
571                         printk(KERN_DEBUG " txsr=%02x gp=%04x rp=%04x", txsr,
                               inw(GP_LOW),inw(RX_LOW));
572
573                 if ((axsr & 0x80) && (txsr & TX_READY) == 0)
574                 {
575                         /*
576                          *      A RECTIFIER : y a-t-il une logique si on garde sur essai ou
577                          *      si on reinitialise immediatement ?
578                          */
579                         if(el_debug>1)
580                                 printk(KERN_DEBUG "%s: Unusual interrupt during Tx,
                                        txsr=%02x axsr=%02x"
581                                         " gp=%03x rp=%03x.\n", dev->name, txsr, axsr,
582                         inw(ioaddr + EL1_DATAPTR), inw(ioaddr + EL1_RXPTR));
583                         lp->txing = 0;
584                         netif_wake_queue(dev);
585                 }
586                 else if (txsr & TX_16COLLISIONS)
587                 {
588                         /*
```

```
589                          *      Delai depasse
590                          */
591                         if (el_debug)
592                                 printk (KERN_DEBUG "%s: Transmit failed 16 times,
                                            Ethernet jammed?\n",dev->name);
593                         outb(AX_SYS, AX_CMD);
594                         lp->txing = 0;
595                         lp->stats.tx_aborted_errors++;
596                         netif_wake_queue(dev);
597                 }
598                 else if (txsr & TX_COLLISION)
599                 {
600                         /*
601                          *      Reenclencher l'emission.
602                          */
603
604                         if (el_debug > 6)
605                                 printk(KERN_DEBUG " retransmitting after a collision.\n");
606                         /*
607                          *      Pauvre petite puce qui ne peut pas reinitialiser son propre
                                    pointeur de debut
608                          */
609
610                         outb(AX_SYS, AX_CMD);
611                         outw(lp->tx_pkt_start, GP_LOW);
612                         outb(AX_XMIT, AX_CMD);
613                         lp->stats.collisions++;
614                         spin_unlock(&lp->lock);
615                         goto out;
616                 }
617                 else
618                 {
619                         /*
620                          *      Ca a marche... nous allons maintenant passer au mode reception
621                          */
622                         lp->stats.tx_packets++;
623                         if (el_debug > 6)
624                                 printk(KERN_DEBUG " Tx succeeded %s\n",
625                                         (txsr & TX_RDY) ? "." : "but tx is busy!");
626                         /*
627                          *      Ceci est sur car l'interruption est atomique a l'egard
                                    d'elle-meme.
628                          */
629
630                         lp->txing = 0;
631                         netif_wake_queue(dev);  /* Au cas ou il y a encore a emettre */
632                 }
633         }
[...]
686 }
```

Autrement dit :

- Le gestionnaire d'interruption récupère le contenu du registre de statut d'émission.
- Si la structure locale de réseau indique qu'on est en train de recevoir des données : si le niveau de débogage est strictement supérieur à 2, on affiche un message noyau ; on force la continuation de la réception des données, on déverrouille la structure locale de réseau et on renvoie `IRQ_HANDLED`.
- Si le niveau de débogage est strictement supérieur à 6, on affiche un message noyau sur le contenu des registres.

– Si l'un des quatre bits forts du registre `ASR` est non nul et si le périphérique est prêt à transmettre : si le niveau de débogage est strictement supérieur à 1, on affiche un message noyau spécifiant qu'il s'agit d'une interruption non usuelle ; on change le champ de la structure locale de réseau et on réveille la file d'attente.

La fonction `netif_wake_queue()` est définie dans le fichier `linux/include/linux/-netdevice.h` :

Code Linux 2.6.10

```
611 static inline void netif_wake_queue(struct net_device *dev)
612 {
613 #ifdef CONFIG_NETPOLL_TRAP
614         if (netpoll_trap())
615                 return;
616 #endif
617         if (test_and_clear_bit(__LINK_STATE_XOFF, &dev->state))
618                 __netif_schedule(dev);
619 }
```

– Si le nombre de 16 collisions est atteint : si le niveau de débogage est non nul, on affiche un message noyau ; on indique à la carte de ne plus réessayer de transmettre les données ; on spécifie à la structure locale de réseau qu'on n'est plus dans l'état d'émission ; on incrémente le nombre d'abandons en transmission dans les informations statistiques de la structure locale de réseau et on réveille la file d'attente.
– Lorsqu'il y a une collision : si le niveau de débogage est strictement supérieur à 6, on affiche un message noyau ; on indique à la carte du périphérique réseau de retransmettre les données ; on met à jour les statistiques de la structure locale de réseau sur le nombre de collisions ; on déverrouille la structure locale de réseau et on renvoie `IRQ_HANDLED`.
– Sinon, c'est qu'on a réussi à transmettre les données : on met à jour les statistiques de la structure locale de réseau sur le nombre de paquets émis ; si le niveau de débogage est strictement supérieur à 6, on affiche un message noyau ; on spécifie à la structure locale de réseau qu'on n'est plus en train de transmettre et on réveille la file d'attente.

34.3.2 Action lorsque le délai est écoulé

34.3.2.1 Fonction indépendante de la carte

Lorsque l'heure système dépasse l'heure `trans_start` du périphérique de la durée du délai `watchdog_timeo`, la couche réseau fait appel à la méthode `tx_timeout()` du pilote de périphérique, comme indiqué par la méthode `dev_watchdog()`, définie dans le fichier `linux/net/sched/-sch_generic.c` :

Code Linux 2.6.10

```
185 static void dev_watchdog(unsigned long arg)
186 {
187         struct net_device *dev = (struct net_device *)arg;
188
189         spin_lock(&dev->xmit_lock);
190         if (dev->qdisc != &noop_qdisc) {
191                 if (netif_device_present(dev) &&
192                     netif_running(dev) &&
193                     netif_carrier_ok(dev)) {
194                         if (netif_queue_stopped(dev) &&
195                             (jiffies - dev->trans_start) > dev->watchdog_timeo) {
196                                 printk(KERN_INFO "NETDEV WATCHDOG: %s: transmit timed out\n",
                                               dev->name);
197                                 dev->tx_timeout(dev);
198                         }
199                         if (!mod_timer(&dev->watchdog_timer, jiffies + dev->watchdog_timeo))
200                                 dev_hold(dev);
```

```
201                 }
202         }
203         spin_unlock(&dev->xmit_lock);
204
205         dev_put(dev);
206 }
```

34.3.2.2 Cas de la carte 3Com 501

Code Linux 2.6.10

Comme nous l'avons vu au chapitre 17, la fonction `tx_timeout()` spécifique à la carte 3Com 501 s'appelle `el_timeout()`. Celle-ci est définie dans le fichier `linux/drivers/net/3c501.c`:

```
363 /**
364  * el_timeout :
365  * @dev : La carte 3c501 dont le delai est ecoule
366  *
367  * Essaie de redemarrer la carte. Fondamentalement, ceci est un melange d'extreme
368  * violence et de priere
369  *
370  */
371
372 static void el_timeout(struct net_device *dev)
373 {
374         struct net_local *lp = netdev_priv(dev);
375         int ioaddr = dev->base_addr;
376
377         if (el_debug)
378                 printk (KERN_DEBUG "%s: transmit timed out, txsr %#2x axsr=%02x rxsr=%02x.\n",
379                         dev->name, inb(TX_STATUS), inb(AX_STATUS), inb(RX_STATUS));
380         lp->stats.tx_errors++;
381         outb(TX_NORM, TX_CMD);
382         outb(RX_NORM, RX_CMD);
383         outb(AX_OFF, AX_CMD);   /* Juste declencher une fausse interruption. */
384         outb(AX_RX, AX_CMD);    /* Controle aux : irq et reception activees  */
385         lp->txing = 0;          /* Revenir a la RX                           */
386         netif_wake_queue(dev);
387 }
```

Autrement dit :

- on déclare une structure de réseau locale, que l'on initialise avec la partie privée du descripteur de périphérique passé en argument ;
- on déclare une adresse d'entrée-sortie, que l'on initialise avec celle fournie par le descripteur de périphérique passé en argument ;
- si `el_debug` est positionné, on affiche un message noyau ;
- on incrémente le nombre d'erreurs dans les informations statistiques de la structure locale de réseau ;
- on redémarre la carte, ce qui est extrêmement violent comme indiqué en commentaire ;
- on réveille la file d'attente.