Chapitre 12

Les tampons de socket

Nous avons vu que chaque paquet (reçu ou à envoyer) occupe une partie de la mémoire vive, avec un enrobage d'en-têtes (et éventuellement d'en-queues) en couches d'oignon. Linux ne réserve pas un emplacement en mémoire vive pour chaque couche, mais un seul emplacement pour le paquet, dont le début peut s'accroître ou rétrécir à souhait pour tenir compte des en-têtes des différentes couches. Il en est de même de la fin pour tenir compte des en-queues des différentes couches. On parle de *tampon* pour cet emplacement, comme pour tout emplacement en mémoire vive, et plus précisément de **tampon de socket** (*socket buffer* en anglais) pour les distinguer, par exemple, des tampons de l'antémémoire des disques.

À chaque tampon de socket est associé un **descripteur de tampon de socket**, permettant de le manipuler.

Ces descripteurs sont placés dans diverses files d'attente.

Dans ce chapitre nous allons décrire la structure des tampons de socket, des descripteurs de tampon de socket et des files d'attente de descripteurs de tampon de socket, c'est-à-dire l'aspect statique. Dans le chapitre suivant nous en étudierons la gestion, c'est-à-dire l'aspect dynamique.

12.1 Structure d'un tampon de socket

Un tampon de socket est un emplacement mémoire sur lequel il n'y a rien à dire *a priori* (contrairement à la structure de son descripteur). Les notions d'espace de tête, d'espace de queue et de fragmentation sont cependant nécessaires pour la suite.

12.1.1 Espace de tête et espace de queue d'un tampon de socket

Un tampon de socket en émission contiendra à un certain moment le paquet à transmettre à la carte réseau. Cependant, durant le déroulement de la pile réseau, ce tampon commence par recevoir les données de l'application, puis l'en-tête de la couche transport, puis l'en-tête de la couche réseau, puis l'en-tête MAC. De même, les données de l'application seront éventuellement suivies d'en-queues divers.

Sous Linux, la zone de mémoire vive non encore remplie, située avant la partie du paquet valide à un moment donné, est appelée **espace de tête** (*headroom* en anglais). De même, la zone de mémoire placée après la partie du paquet valide à cet instant est appelée **espace de queue** (**tailroom** en anglais).

12.1.2 Fragmentation d'un tampon de socket

Les tampons de socket peuvent être trop grand pour tenir dans une page. Dans ce cas ils sont dispersés sur plusieurs pages, on dit qu'ils sont **fragmentés**. De plus nous avons vu qu'il peut s'agir du regroupement de plusieurs tampons ; il est plus simple dans ce cas de conserver cette structure marquant le regoupement (tout au moins momentanément) plutôt que de "linéariser" immédiatement.

12.1.2.1 Descripteur de fragment

Un fragment de tampon de socket sera contenu entièrement dans une page. Cependant il n'occupe pas nécessairement la page en entier, ne serait-ce que parce que la taille du tampon n'est pas un multiple de la taille d'une page.

Le type `struct skb_frag_t` permet de détenir les informations sur les fragments. Il est défini dans le fichier en-tête `linux/include/linux/skbuff.h`:

Code Linux 2.6.10

```
129 typedef struct skb_frag_struct skb_frag_t;
130
131 struct skb_frag_struct {
132         struct page *page;
133         __u16 page_offset;
134         __u16 size;
135 };
```

qui spécifie le numéro de page, le **décalage** ou adresse du début des données dans cette page et la taille, c'est-à-dire le nombre d'octets utiles dans cette page à partir du décalage. Ceci détermine une zone contiguë de mémoire, entièrement contenue dans une même page.

12.1.2.2 Informations sur la fragmentation

Les informations sur la fragmentation font l'objet d'une entité du type structuré `struct skb_shared_info`, qui comprend un tableau de fragments et une liste de descripteurs de tampon de socket. Celui-ci est défini dans le fichier en-tête `linux/include/linux/skbuff.h`:

Code Linux 2.6.10

```
137 /* Ces donnees ne varient pas d'un clone a l'autre et sont situees a la
138  * fin de l'en-tete des donnees, c'est-a-dire a skb->end.
```

```
139  */
140 struct skb_shared_info {
141         atomic_t        dataref;
142         unsigned int    nr_frags;
143         unsigned short  tso_size;
144         unsigned short  tso_segs;
145         struct sk_buff  *frag_list;
146         skb_frag_t      frags[MAX_SKB_FRAGS];
147 };
```

dont les champs spécifient :

- s'il est effectivement fait référence à cette entité ou non (autrement dit si on peut l'effacer de la mémoire vive si besoin est) ;
- le nombre de fragments dans le tableau de fragments, autrement dit la taille effective de celui-ci ;
- la taille et les segments concernat le protocole de transport TCP (avec `tso` pour *TCP Segmentation On*) ;
- une **liste de fragments**, chaque fragment étant un (descripteur de) tampon de socket ;
- un **tableau de fragments** représentant des emplacements en mémoire vive permettant de détenir 64 Ko de données sans avoir à recourir à une liste de fragments (champ précédent).

  Le nombre `MAX_SKB_FRAGS` de pages nécessaires pour détenir 64 Ko est également défini dans le fichier en-tête `linux/include/linux/skbuff.h`:

Code Linux 2.6.10

```
126 /* Pour permettre une trame de 64K d'etre empaquetee comme un seul skb sans frag_list */
127 #define MAX_SKB_FRAGS (65536/PAGE_SIZE + 2)
```

Remarque

Ordre de citation du code.- Nous nous arrangerons toujours, lors de la citation du code de Linux, pour ne pas avoir de référence en avant, c'est-à-dire à ne pas utiliser de type ou de fonction qui ne serait étudié que plus loin.

Lors de la définition du type `struct skb_frag_t`, on rencontre le type `__u16`, mais celui-ci a déjà été introduit précédemment. Il est vrai que l'on rencontre également le type `page`. Il s'agit d'un type général concernant la gestion de la mémoire. Nous ne pouvons pas revoir tout le noyau Linux.

On rencontre également, lors de la définition du type `struct skb_shared_info`, *a priori* le type `sk_buff` non défini antérieurement. En fait si on observe bien, ce n'est pas ce type qui intervient mais un pointeur sur ce type, c'est-à-dire une adresse. On pourrait tout aussi bien le remplacer par le type `void *`. On n'a donc pas besoin de connaître explicitement ce type à ce moment. Cette situation se retrouvera très souvent dans la suite.

12.2 Descripteur de tampon

Linux définit un type de descripteur de tampon de socket assez complexe, différent des autres implémentations dont les sources sont disponibles.

La figure 10-1 ([WPRMB–02], p. 66) montre les relations entre un tampon et son descripteur.

Commençons par citer la définition du type descripteur de tampon de socket avant d'en commenter les champs un à un dans les sections suivantes.

12.2.1 Définition du type

Le type `struct sk_buff` est défini dans le fichier `linux/include/linux/skbuff.h`:

Code Linux 2.6.10

```
1 /*
```

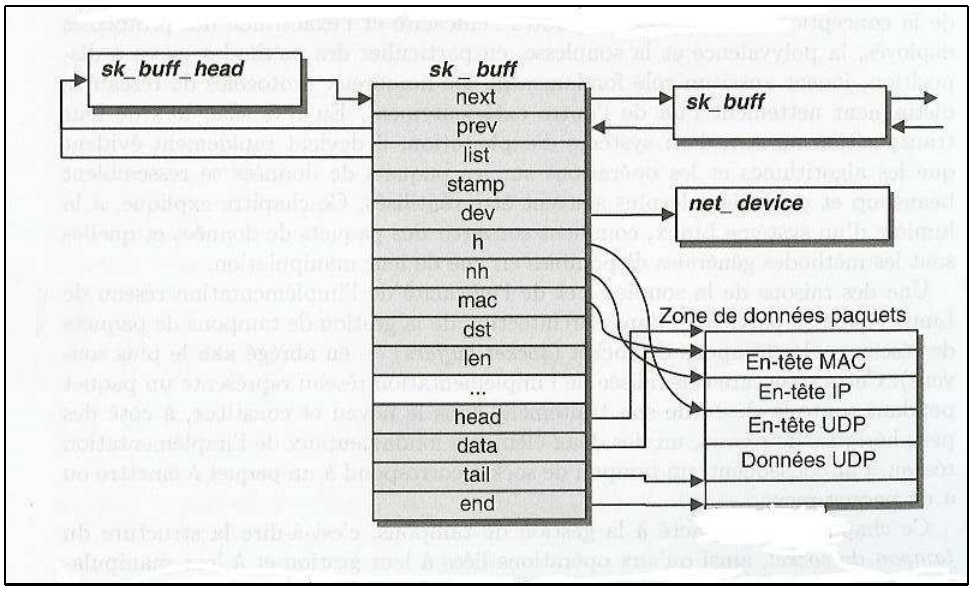

FIG. 12.1 – *Tampon de socket et son descripteur*

```
2   *      Definitions pour les routines de memoire de 'struct sk_buff'.
3   *
4   *      Auteurs :
5   *              Alan Cox, <gw4pts@gw4pts.ampr.org>
6   *              Florian La Roche, <rzsfl@rz.uni-sb.de>
[...]
14 #ifndef _LINUX_SKBUFF_H
15 #define _LINUX_SKBUFF_H
[...]
149 /**
150  *      struct sk_buff - tampon de socket
151  *      @next : Tampon suivant dans la liste
152  *      @prev : Tampon precedent dans la liste
153  *      @list : Liste dans laquelle nous nous trouvons
154  *      @sk : Socket dont nous sommes la propriete
155  *      @stamp : Heure a laquelle nous sommes arrives
156  *      @dev : Peripherique par lequel nous sommes arrives/nous partirons
157  *      @input_dev : Peripherique par lequel nous sommes arrives
158  *      @real_dev : Le peripherique effectif que nous sommes en train d'utiliser
159  *      @h : En-tete de la couche transport
160  *      @nh : En-tete de la couche reseau
161  *      @mac : En-tete de la couche liaison
162  *      @dst : A RECTIFIER : Decrire ce champ
163  *      @cb : Tampon de controle. Utilisation libre par chaque couche. Mettre les vars
        privees ici
164  *      @len : Longueur des donnees en cours
165  *      @data_len : Longueur des donnees
166  *      @mac_len : Longueur de l'en-tete de couche de liaison
167  *      @csum : Somme de controle
168  *      @__unused : Champ decede, peut etre reutilise
169  *      @cloned : La tete peut etre clonee (verifier refcnt pour en etre sur)
170  *      @pkt_type : Classe de paquet
171  *      @ip_summed : Le peripherique nous a fourni une somme de controle IP
172  *      @priority : Priorite de mise en file d'attente du paquet
```

```
173  *      @users : Compteur d'utilisateurs - voir {datagram,tcp}.c
174  *      @protocol : Protocole du paquet depuis le peripherique
175  *      @security : Niveau de securite du paquet
176  *      @truesize : Taille du tampon
177  *      @head : Tete du tampon
178  *      @data : Pointeur sur le debut des donnees
179  *      @tail : Pointeur sur la fin des donnees
180  *      @end : Pointeur sur la fin
181  *      @destructor : Fonction de destruction
182  *      @nfmark : Peut etre utilise pour la communication entre points d'ancrage
183  *      @nfcache : Infos sur le cache
184  *      @nfct : Connexion associee, si elle existe
185  *      @nfctinfo : Relations de ce skb avec la connexion
186  *      @nf_debug : Debogage Netfilter
187  *      @nf_bridge : Donnees sauvegardees sur une trame de pont - voir br_netfilter.c
188  *      @private : Donnees qui sont privees pour l'implementation de HIPPI
189  *      @tc_index : Index de controle du trafic
190  */
191
192 struct sk_buff {
193         /* Ces deux membres doivent apparaitre en premier. */
194         struct sk_buff          *next;
195         struct sk_buff          *prev;
196
197         struct sk_buff_head     *list;
198         struct sock             *sk;
199         struct timeval          stamp;
200         struct net_device       *dev;
201         struct net_device       *input_dev;
202         struct net_device       *real_dev;
203
204         union {
205                 struct tcphdr   *th;
206                 struct udphdr   *uh;
207                 struct icmphdr  *icmph;
208                 struct igmphdr  *igmph;
209                 struct iphdr    *ipiph;
210                 struct ipv6hdr  *ipv6h;
211                 unsigned char   *raw;
212         } h;
213
214         union {
215                 struct iphdr    *iph;
216                 struct ipv6hdr  *ipv6h;
217                 struct arphdr   *arph;
218                 unsigned char   *raw;
219         } nh;
220
221         union {
222                 unsigned char   *raw;
223         } mac;
224
225         struct  dst_entry       *dst;
226         struct  sec_path        *sp;
227
228         /*
229          * Ceci est le tampon de controle. Son utilisation par chaque couche est
230          * libre. Mettre vos variables privees ici s'il vous plait. Si vous
231          * voulez les conserver en passant d'une couche a une autre, vous devez d'abord
232          * faire un skb_clone(). Ce tampon est possede par celui qui a mis le skb dans la file
              d'attente ATM.
233          */
```

```
234         char                    cb[40];
235
236         unsigned int            len,
237                                 data_len,
238                                 mac_len,
239                                 csum;
240         unsigned char           local_df,
241                                 cloned,
242                                 pkt_type,
243                                 ip_summed;
244         __u32                   priority;
245         unsigned short          protocol,
246                                 security;
247
248         void                    (*destructor)(struct sk_buff *skb);
249 #ifdef CONFIG_NETFILTER
250         unsigned long           nfmark;
251         __u32                   nfcache;
252         __u32                   nfctinfo;
253         struct nf_conntrack     *nfct;
254 #ifdef CONFIG_NETFILTER_DEBUG
255         unsigned int            nf_debug;
256 #endif
257 #ifdef CONFIG_BRIDGE_NETFILTER
258         struct nf_bridge_info   *nf_bridge;
259 #endif
260 #endif /* CONFIG_NETFILTER */
261 #if defined(CONFIG_HIPPI)
262         union {
263                 __u32           ifield;
264         } private;
265 #endif
266 #ifdef CONFIG_NET_SCHED
267         __u32                   tc_index;       /* index de controle du trafic */
268 #ifdef CONFIG_NET_CLS_ACT
269         __u32           tc_verd;                /* verdict de controle de trafic */
270         __u32           tc_classid;             /* id de classe de controle de trafic */
271 #endif
272
273 #endif
274
275
276         /* Ces elements doivent etre a la fin, voir alloc_skb() pour les details.  */
277         unsigned int            truesize;
278         atomic_t                users;
279         unsigned char           *head,
280                                 *data,
281                                 *tail,
282                                 *end;
283 };
```

12.2.2 Champs relatifs aux listes de descripteurs de tampon

12.2.2.1 Parcours de la liste de descripteurs de tampon

Les deux premiers champs `next` et `prev` permettent au descripteur de tampon de se situer dans une liste doublement chaînée.

12.2.2.2 File d'attente à laquelle appartient le tampon

Le champ `list` indique la file d'attente dans laquelle le descripteur de tampon se trouve en ce moment. Les files d'attente doivent par conséquent toujours être du type `struct sk_buff_head`, afin que les opérations de tampons de socket puissent les gérer. Si un tampon n'est assigné à aucune file d'attente, ce champ prend la valeur `NULL`. Nous reviendrons dans la troisième section sur la structure des files d'attente de descripteurs de tampon de socket.

12.2.3 Provenance du tampon

La provenance du tampon est indiquée par plusieurs champs :

- L'attribut `sk` indique le descripteur de couche transport associé au tampon, qu'il soit un tampon d'émission ou de réception. Le type `struct sock`, dont nous n'avons pas vraiment besoin ici (seul le type adresse est en fait utilisé), sera étudié au moment opportun, c'est-à-dire au chapitre 28.
- L'attribut `stamp` indique le moment précis, en `jiffies`, où il est arrivé (de l'extérieur en cas de réception, depuis le processus en cas d'émission).
- L'attribut `dev` spécifie le périphérique réseau auquel il est attaché : depuis lequel il provient en cas de réception, auquel il est destiné en cas d'émission. Le type `struct net_device`, dont nous n'avons pas vraiment besoin ici (seul le type adresse est en fait utilisé), sera étudié au moment opportun.
- Dans le cas d'un routeur (ce qui ne nous intéressera pas dans cet ouvrage), l'attribut `input_dev` spécifie le périphérique réseau duquel il provient, l'attribut précédent le périphérique par lequel il partira.
- L'attribut `real_dev` spécifie le périphérique réseau que nous sommes en train d'utiliser, qui peut être différent des périphériques ci-dessus.

12.2.4 En-têtes des couches de protocole

Nous avons vu que les tampons de socket contiennent les données de l'application ainsi que les encapsulations successives. Les trois champs suivants permettent donc de préciser les en-têtes de la couche transport, de la couche réseau et de la sous-couche MAC.

12.2.4.1 En-tête de la couche de transport

L'en-tête `h` (pour *Header*) de la couche de transport dépend évidemment du protocole de transport. Le type associé est donc une union, dont l'élément est l'en-tête des protocoles TCP (type `tcphdr`), UDP (type `udphdr`), ICMP (type `icmphdr`), IGM (type `igmhdr`, défini dans le fichier en-tête `linux/include/linux/igm.h`), IPv4 (type `iphdr`, qui concerne normalement la couche réseau et non la couche de transport sauf dans le cas de données brutes), IPv6 (type `ipv6hdr`, défini dans le fichier en-tête `linux/include/linux/ipv6.h`) ou brut (qui est alors tout simplement une chaîne de caractères). La structure de chacun de ces en-têtes, qui n'est pas utile ici (seul le type adresse est utilisé), sera décrite aux chapitres consacrés à ces protocoles.

12.2.4.2 En-tête de la couche réseau

L'en-tête `nh` (pour *Net Header*) de la couche réseau dépend lui aussi du protocole choisi, protocole de réseau dans ce cas. Le type associé est donc également une union, dont l'élément est l'en-tête des protocoles IP version 4 (type `iphdr`), IP version 6, ARP (type `arphdr` défini dans le fichier en-tête `linux/include/if_arp.h`) ou brut (qui est alors tout simplement une chaîne de

caractères). Comme pour le champ précédent, la structure de chacun de ces en-têtes, qui n'est pas utile ici, sera décrite aux chapitres consacrés à ces protocoles.

12.2.4.3 En-tête de la sous-couche MAC

L'en-tête `mac` de la sous-couche MAC dépend lui aussi du protocole choisi. Le type associé est donc également une union, dont l'élément a été longtemps soit un en-tête Ethernet (type `ethhdr`) soit brut (qui est alors tout simplement une chaîne de caractères). On n'a plus le choix depuis la version 2.6.10, mais on a gardé l'union pour la compatibilité avec les versions antérieures.

12.2.5 Attributs divers

12.2.5.1 Entrée de cache de destination

Le champ `dst` renvoie à une entrée dans le cache de routage. Il contient des informations sur l'itinéraire du paquet, par exemple la carte réseau sur laquelle le paquet doit quitter l'ordinateur (dans le cas où il y en a plusieurs) et éventuellement un renvoi sur un en-tête MAC tout fait déjà stocké comme champ de cette entrée. Nous étudierons le type `dst_entry`, qui n'est pas vraiment utile ici, avec les structures de données pour IP.

12.2.5.2 Chemin sécurisé

Le champ `sp` est utile lorsqu'on veut un chemin sécurisé, ce qui ne nous intéressera pas dans ce livre. Le type `struct sec_path` n'est pas vraiment utile, puisqu'il s'agit d'une adresse.

12.2.5.3 Bloc de contrôle du protocole

Le champ `cb` contient des informations concernant la couche que l'on est en train de traiter et à usage libre par celle-ci. Ces informations sont surchargées lors du passage d'une couche à une autre, d'où l'avertissement concernant le clonage s'il est nécessaire de ne pas pas perdre celles-ci.

12.2.5.4 Attributs généraux

Les attributs généraux sont les suivants :

- La taille `len` désigne la longueur de la partie du tampon qui est renseignée à ce moment. Il s'agit d'un attribut redondant puisque ce n'est rien d'autre que `tail - data`. Seules les données accessibles par le noyau sont prises en considération. Avec une trame Ethernet par exemple, seules les deux adresses MAC et le champ type/longueur sont pris en considération. Les autres champs (préambule, champ de remplissage et somme de contrôle) sont traités par la carte réseau, par conséquent le noyau n'en tient pas compte.
- La taille `data_len` désigne la longueur de la partie du paquet qui a un sens lors du traitement dans la couche dans laquelle nous nous trouvons.
- Le champ `mac_len` spécifie la longueur de l'en-tête de la sous-couche MAC de la couche liaison.
- Le champ `csum` spécifie la somme de contrôle.
- Le champ `local_df` (auparavant `__unused`, comme encore référencé dans le commentaire) est à usage local.

– Le champ `cloned` indique si le descripteur de tampon a été cloné. Si c'est le cas, on se trouve alors en présence de plusieurs descripteurs de tampon faisant référence au même tampon de socket.

– Le type ou classe du paquet `pkt_type` est utilisé lors de la livraison du paquet. Le pilote doit lui assigner l'une des valeurs `PACKET_HOST` (paquet à remettre localement), `PACKET_BROADCAST` (paquet de diffusion générale), `PACKET_MULTICAST` (paquet de multidiffusion), `PACKET_OTHERHOST` (paquet qui n'était pas destiné à l'ordinateur local, mais dont la réception a été rendue possible grâce au un mode spécial de promiscuité), `PACKET_OUTGOING` (paquet devant quitter l'ordinateur), `PACKET_LOOPBACK` (paquet provenant du système local vers lui-même) ou `PACKET_FASTROUTE` (transmission rapide entre cartes réseau spéciales).

Les constantes symboliques représentant les types de paquets sont définies dans le fichier en-tête `linux/include/linux/if_packet.h`: Code Linux 2.6.10

```
22 /* Types de paquet */
23
24 #define PACKET_HOST             0               /* Pour nous                */
25 #define PACKET_BROADCAST        1               /* Pour tous                */
26 #define PACKET_MULTICAST        2               /* Pour le groupe           */
27 #define PACKET_OTHERHOST        3               /* Pour quelqu'un d'autre   */
28 #define PACKET_OUTGOING         4               /* Sortie de tout type      */
29 /* Ceux-ci sont invisibles au niveau utilisateur */
30 #define PACKET_LOOPBACK         5               /* MC/BRD frame looped back */
31 #define PACKET_FASTROUTE        6               /* Trame Fastroutee         */
```

– Le comportement `ip_summed` du pilote de périphérique à l'égard de la somme de contrôle IP est représenté par l'une des des constante symboliques suivantes (aucune, fournie par le matériel ou non nécessaire), définies dans le fichier `linux/include/linux/skbuff.h`: Code Linux 2.6.10

```
36 #define CHECKSUM_NONE        0
37 #define CHECKSUM_HW          1
38 #define CHECKSUM_UNNECESSARY 2
```

Un commentaire se trouve dans le fichier en-tête `linux/include/linux/skbuff.h`: Code Linux 2.6.10

```
48 /* A. Verification de la somme des paquets recus par le peripherique.
49  *
50  *      NONE : le peripherique a echoue en verifiant la somme de ce paquet.
51  *              skb->csum n'est pas defini.
52  *
53  *      UNNECESSARY : le peripherique a etudie le paquet et a du verifie la somme.
54  *              skb->csum n'est pas defini.
55  *            C'est une mauvaise option, mais malheureusement beaucoup de vendeurs font
56  *            ceci. Apparemment dans le but secret de vous vendre un nouveau peripherique
57  *            lorsque vous ajouterez un nouveau protocole a votre hote, par exemple
                  IPv6. 8)
58  *
59  *      HW : La facon la plus generique. Le peripherique supplee la somme de controle de
60  *          _tout_ le paquet comme vu par netif_rx dans skb->csum.
61  *          NOTE : Meme si le peripherique supporte seulement quelques protocoles, mais
62  *          est capable de produire quelques skb->csum, il DOIT utiliser HW,
63  *          et non UNNECESSARY.
64  *
65  * B. Verification en sortie.
66  *
67  *      NONE : la somme de controle de skb est calculee par le protocole ou csum n'est
                  pas necessaire.
68  *
69  *      HW : le peripherique est necessaire pour calculer la somme csum du paquet comme
70  *      vu par hard_start_xmit a partir de skb->h.raw a la fin et pour enregister la
71  *      somme de controle a skb->h.raw+skb->csum.
```

Il est un peu curieux de rencontrer ainsi un champ relatif à une couche donnée, et même à un seul protocole de cette couche, mais c'est ainsi.

- Le champ `priority` spécifie la priorité du paquet.
- Le champ `protocol` sert à transmettre le tampon de la sous-couche LLC à la couche réseau. La valeur qui nous intéresse principalement est `ETH_P_IP`. Les valeurs possibles seront définies lors de l'étude de l'implémentation du protocole Ethernet.
- Le champ `security` spécifie le niveau de sécurité du paquet, ce qui ne nous intéressera pas dans cet ouvrage.

12.2.5.5 Fonction de destruction du tampon

Le tampon devra être détruit après utilisation. La façon de faire dépend d'un certain nombre de paramètres, d'où l'intérêt de la fonction membre `destructor()`.

12.2.5.6 Attributs optionnels

Un certain nombre d'attributs complémentaires existent suivant les options choisies pour la compilation du noyau, dont aucune ne nous intéressera dans cet ouvrage :

- `nfmark` pour la communication entre points d'ancrage si on a choisi le filtrage réseau ;
- `nfcache` pour des informations sur le cache si on a choisi le filtrage réseau ;
- `nfctinfo` pour des informations sur le cache si on a choisi le filtrage réseau ;
- `nfct` pour la connexion associée si on a choisi le filtrage réseau ;
- `nf_debug` si on a choisi le filtrage réseau avec option de débogage ;
- `nf_bridge` pour sauvegarder les données d'une trame dans le cas d'un pont ;
- `private` si on a choisi l'option des périphériques à haut débit `HIPPI` (*High-Performance Parallel Interface*) ;
- l'index de contrôle du trafic si on a choisi l'option `CONFIG_NET_SCHED`.
- et enfin le verdict et l'identificateur de classe de contrôle du trafic si on a choisi l'option `CONFIG_NET_CLS_ACT`.

12.2.5.7 Attributs finaux

Les derniers attributs doivent être placés à la fin pour faciliter l'implémentation de la fonction `alloc_skb()`. Ils concernent :

- La taille `truesize` du tampon, données, en-têtes et attributs de contrôle compris.
- Le nombre `users` d'utilisateurs du tampon.
- L'entreposage du tampon. L'emplacement du tampon en mémoire vive est caractérisé par quatre attributs :
  - un pointeur `head` sur le début du tampon (espace alloué), autrement dit le début de l'espace de tête (*headroom*) ;
  - un pointeur `data` sur le début des données remplies de façon valide à ce moment, dont l'adresse est donc supérieure à l'adresse précédente ;
  - un pointeur `tail` sur l'octet suivant la fin des données remplies de façon valide à ce moment, autrement dit le début de l'espace de queue (*tailroom*) ;
  - un pointeur `end` sur l'adresse de fin de l'espace alloué, supérieur ou égal à `tail`.

12.2.5.8 Informations sur la fragmentation du tampon

Une entité du type `struct skb_shared_info` est repérée par le champ `end`, qui joue donc deux rôles.

12.3 Files d'attente de tampons de socket

Lorsqu'un paquet n'est pas traité immédiatement, il est placé dans une file d'attente. Linux prend en charge la gestion des paquets grâce à la structure de file d'attente de descripteurs de socket `sk_buff_head` et à des opérations afférentes.

Comme le montre la figure 12-2 ([WPRMB–02], p.79), les descripteurs de tampon de socket stockés dans une file d'attente sont doublement chaînés l'un à l'autre au moyen d'une structure en anneau. Le double chaînage permet une navigation rapide dans les deux sens. La structure en anneau facilite le chaînage et évite l'apparition du pointeur `NULL`.

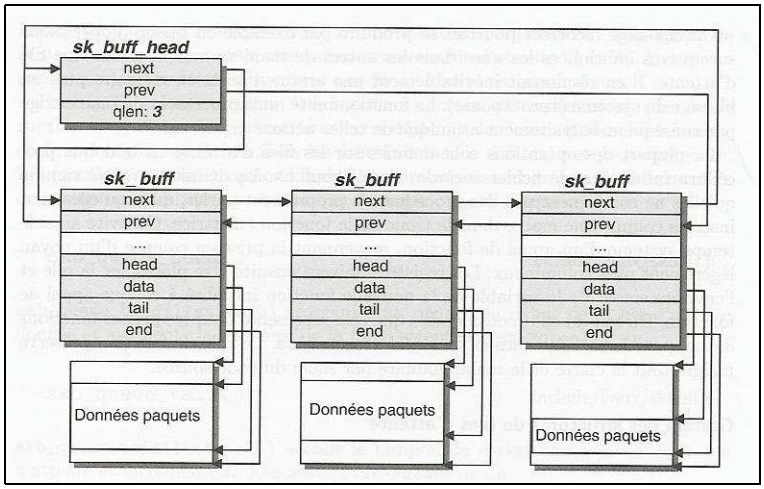

FIG. 12.2 – *File d'attente de paquets*

Le type `sk_buff_head` est défini dans le fichier en-tête `linux/include/linux/skbuff.h`: Code Linux 2.6.10

```
115 struct sk_buff_head {
116         /* Ces deux membres doivent etre en premier. */
117         struct sk_buff  *next;
118         struct sk_buff  *prev;
119
120         __u32           qlen;
121         spinlock_t      lock;
122 };
```

- les champs `next` et `prev` des descripteurs de tampon de socket servent au chaînage des descripteurs; le champ `next` de `sk_buff_head` indique le premier paquet placé dans la file d'attente et `prev` le dernier;
- le champ `qlen` (pour l'anglais *Queue LENgth*) indique le nombre d'éléments présents dans la file d'attente;
- le champ `lock` est un verrou rotatif utilisé lors de l'exécution d'opérations atomique sur la file d'attente. Dans ce cas, lors d'un accès critique, le verrou rotatif doit être libre ou attendu jusqu'à sa libération.