Chapitre 25

Réception des datagrammes UDP

Après une étude générale de l'implémentation Linux de la couche de transport UDP (en-tête et calcul de la somme de contrôle), on étudie la réception des datagrammes dans la couche de transport UDP.

25.1 Implémentation Linux générale d'UDP

25.1.1 Implémentation Linux de l'en-tête UDP

L'en-tête UDP est une entité du type `struct udphdr`. Celui-ci est défini dans le fichier en-tête `linux/include/linux/udp.h`:

Code Linux 2.6.10

```
6   *              Definitions pour le protocole UDP.
7   *
8   * Version :    @(#)udp.h       1.0.2   04/28/93
9   *
10  * Auteur :     Fred N. van Kempen, <waltje@uWalt.NL.Mugnet.ORG>
[...]
17 #ifndef _LINUX_UDP_H
18 #define _LINUX_UDP_H
19
20 #include <linux/types.h>
21
22 struct udphdr {
23         __u16   source;
24         __u16   dest;
25         __u16   len;
26         __u16   check;
27 };
```

25.1.2 Calcul de la somme de contrôle

25.1.2.1 Calcul partiel de la somme de contrôle d'un bloc de mémoire

La fonction `csum_partial()` permet de calculer la somme de contrôle d'un bloc de mémoire. Elle dépend du microprocesseur. Elle est déclarée dans le fichier `linux/include/asm-i386/checksum.h` dans le cas des microprocesseurs Intel :

Code Linux 2.6.10

```
6  /*
7   * Calcule la somme de controle d'un bloc de memoire d'un buff, de longueur len,
8   * et l'ajoute a "sum" (32-bit)
9   *
10  * Renvoie un nombre de 32 bits adequat en lui-meme
11  * ou pour csum_tcpudp_magic
12  *
13  * Cette fonction doit etre appelee avec des longueurs paires, sauf pour
14  * le dernier fragment, dont la longueur peut etre impaire
15  *
16  * C'est mieux d'avoir des buff alignes sur 32 bits
17  */
18 asmlinkage unsigned int csum_partial(const unsigned char * buff, int len,
                                      unsigned int sum);
```

La définition de cette fonction, en langage d'assemblage, se trouve dans le fichier `linux/arch/i386/lib/checksum.S`:

Code Linux 2.6.10

```
6    *              Routines de calcul de la somme de controle pour IP/TCP/UDP
7    *
8    * Auteurs :    Jorge Cwik, <jorge@laser.satlink.net>
9    *              Arnt Gulbrandsen, <agulbra@nvg.unit.no>
10   *              Tom May, <ftom@netcom.com>
11   *              Routines Pentium Pro/II :
12   *              Alexander Kjeldaas <astor@guardian.no>
13   *              Finn Arne Gangstad <finnag@guardian.no>
14   *              Beaucoup de code a ete deplace depuis tcp.c et ip.c ; voir ces fichiers
15   *              pour plus de noms.
16   *
17   * Changements : Ingo Molnar, a converti csum_partial_copy() en routine d'exception
18   *                           2.1.
19   *              Andi Kleen, a ajoute la mise a zero en cas d'erreur
20   *                   a converti en langage d'assemblage pur
[...]
28  #include <linux/config.h>
29  #include <asm/errno.h>
30
31  /*
32   * calcule une somme de controle partielle, par exemple pour des fragments TCP/UDP
33   */
34
35   /*
36  unsigned int csum_partial(const unsigned char * buff, int len, unsigned int sum)
37   */
38
39  .text
40  .align 4
41  .globl csum_partial
42
43  #ifndef CONFIG_X86_USE_PPRO_CHECKSUM
44
45            /*
46             * Les experiences avec les connexions Ethernet et SLIP montrent que buff
47             * est aligne sur une frontiere 2 octets ou 4 octets. Nous obtiendrons au
48             * moins une acceleration double sur 486 et Pentium si c'est aligne sur 4 octets.
```

```
49              * Heureusement, il est facile de convertir un alignement 2 octets en un
50              * alignement 4 octets pour la boucle non enroulee.
51              */
52  csum_partial:
53          pushl %esi
54          pushl %ebx
55          movl 20(%esp),%eax      # Arg de fonction : unsigned int sum
56          movl 16(%esp),%ecx      # Arg de fonction : int len
57          movl 12(%esp),%esi      # Arg de fonction : unsigned char *buff
58          testl $3, %esi          # Verifie l'alignement.
59          jz 2f                   # Saut si l'alignement est ok.
60          testl $1, %esi          # Verifie l'alignement.
61          jz 10f                  # Saut si l'alignement est une frontiere de 2 octets.
62
63          # buf est impair
64          dec %ecx
65          jl 8f
66          movzbl (%esi), %ebx
67          adcl %ebx, %eax
68          roll $8, %eax
69          inc %esi
70          testl $2, %esi
71          jz 2f
72  10:
73          subl $2, %ecx           # L'alignement utilise jusqu'a deux octets.
74          jae 1f                  # Saut si nous avons au moins deux octets.
75          addl $2, %ecx           # ecx etait < 2. Faire avec ca.
76          jmp 4f
77  1:      movw (%esi), %bx
78          addl $2, %esi
79          addw %bx, %ax
80          adcl $0, %eax
81  2:
82          movl %ecx, %edx
83          shrl $5, %ecx
84          jz 2f
85          testl %esi, %esi
86  1:      movl (%esi), %ebx
87          adcl %ebx, %eax
88          movl 4(%esi), %ebx
89          adcl %ebx, %eax
90          movl 8(%esi), %ebx
91          adcl %ebx, %eax
92          movl 12(%esi), %ebx
93          adcl %ebx, %eax
94          movl 16(%esi), %ebx
95          adcl %ebx, %eax
96          movl 20(%esi), %ebx
97          adcl %ebx, %eax
98          movl 24(%esi), %ebx
99          adcl %ebx, %eax
100         movl 28(%esi), %ebx
101         adcl %ebx, %eax
102         lea 32(%esi), %esi
103         dec %ecx
104         jne 1b
105         adcl $0, %eax
106 2:      movl %edx, %ecx
107         andl $0x1c, %edx
108         je 4f
109         shrl $2, %edx           # Ceci met CF a zero
110 3:      adcl (%esi), %eax
```

```
111         lea 4(%esi), %esi
112         dec %edx
113         jne 3b
114         adcl $0, %eax
115 4:      andl $3, %ecx
116         jz 7f
117         cmpl $2, %ecx
118         jb 5f
119         movw (%esi),%cx
120         leal 2(%esi),%esi
121         je 6f
122         shll $16,%ecx
123 5:      movb (%esi),%cl
124 6:      addl %ecx,%eax
125         adcl $0, %eax
126 7:
127         testl $1, 12(%esp)
128         jz 8f
129         roll $8, %eax
130 8:
131         popl %ebx
132         popl %esi
133         ret
134
135 #else
136
137 /* Version pour PentiumII/PPro */
138
139 csum_partial:
140         pushl %esi
141         pushl %ebx
142         movl 20(%esp),%eax      # Arg de fonction : unsigned int sum
143         movl 16(%esp),%ecx      # Arg de fonction : int len
144         movl 12(%esp),%esi      # Arg de fonction : const unsigned char *buf
145
146         testl $3, %esi
147         jnz 25f
148 10:
149         movl %ecx, %edx
150         movl %ecx, %ebx
151         andl $0x7c, %ebx
152         shrl $7, %ecx
153         addl %ebx,%esi
154         shrl $2, %ebx
155         negl %ebx
156         lea 45f(%ebx,%ebx,2), %ebx
157         testl %esi, %esi
158         jmp *%ebx
159
160         # Manipule les regions alignees sur 2 octets
161 20:      addw (%esi), %ax
162         lea 2(%esi), %esi
163         adcl $0, %eax
164         jmp 10b
165 25:
166         testl $1, %esi
167         jz 30f
168         # buf is odd
169         dec %ecx
170         jl 90f
171         movzbl (%esi), %ebx
172         addl %ebx, %eax
```

```
173         adcl $0, %eax
174         roll $8, %eax
175         inc %esi
176         testl $2, %esi
177         jz 10b
178
179 30:     subl $2, %ecx
180         ja 20b
181         je 32f
182         addl $2, %ecx
183         jz 80f
184         movzbl (%esi),%ebx      # calcule la somme 1 octet, 2-aligne
185         addl %ebx, %eax
186         adcl $0, %eax
187         jmp 80f
188 32:
189         addw (%esi), %ax        # calcule la somme 2 octets, 2-aligne
190         adcl $0, %eax
191         jmp 80f
192
193 40:
194         addl -128(%esi), %eax
195         adcl -124(%esi), %eax
196         adcl -120(%esi), %eax
197         adcl -116(%esi), %eax
198         adcl -112(%esi), %eax
199         adcl -108(%esi), %eax
200         adcl -104(%esi), %eax
201         adcl -100(%esi), %eax
202         adcl -96(%esi), %eax
203         adcl -92(%esi), %eax
204         adcl -88(%esi), %eax
205         adcl -84(%esi), %eax
206         adcl -80(%esi), %eax
207         adcl -76(%esi), %eax
208         adcl -72(%esi), %eax
209         adcl -68(%esi), %eax
210         adcl -64(%esi), %eax
211         adcl -60(%esi), %eax
212         adcl -56(%esi), %eax
213         adcl -52(%esi), %eax
214         adcl -48(%esi), %eax
215         adcl -44(%esi), %eax
216         adcl -40(%esi), %eax
217         adcl -36(%esi), %eax
218         adcl -32(%esi), %eax
219         adcl -28(%esi), %eax
220         adcl -24(%esi), %eax
221         adcl -20(%esi), %eax
222         adcl -16(%esi), %eax
223         adcl -12(%esi), %eax
224         adcl -8(%esi), %eax
225         adcl -4(%esi), %eax
226 45:
227         lea 128(%esi), %esi
228         adcl $0, %eax
229         dec %ecx
230         jge 40b
231         movl %edx, %ecx
232 50:     andl $3, %ecx
233         jz 80f
234
```

```
235         # Manipule les derniers 1-3 octets sans saut
236         notl %ecx               # 1->2, 2->1, 3->0, les bits plus hauts sont masques
237         movl $0xffffff,%ebx     # par les instructions shll et shrl
238         shll $3,%ecx
239         shrl %cl,%ebx
240         andl -128(%esi),%ebx    # esi est 4-aligne donc devrait etre ok
241         addl %ebx,%eax
242         adcl $0,%eax
243 80:
244         testl $1, 12(%esp)
245         jz 90f
246         roll $8, %eax
247 90:
248         popl %ebx
249         popl %esi
250         ret
251
252 #endif
```

25.1.2.2 Calcul complet de la somme de contrôle

La fonction `skb_checksum(skb, offset, len, csum)` calcule la somme de contrôle `csum` de la partie des données du tampon `skb` qui commence au décalage `offset` et de longueur `len`. Elle est définie dans le fichier `linux/net/core/skbuff.c`:

Code Linux 2.6.10

```
996  /* Calcul de la somme de controle des donnees skb. */
997
998  unsigned int skb_checksum(const struct sk_buff *skb, int offset,
999                            int len, unsigned int csum)
1000 {
1001         int start = skb_headlen(skb);
1002         int i, copy = start - offset;
1003         int pos = 0;
1004
1005         /* Calcul de la somme de controle de l'en-tete. */
1006         if (copy > 0) {
1007                 if (copy > len)
1008                         copy = len;
1009                 csum = csum_partial(skb->data + offset, copy, csum);
1010                 if ((len -= copy) == 0)
1011                         return csum;
1012                 offset += copy;
1013                 pos     = copy;
1014         }
1015
1016         for (i = 0; i < skb_shinfo(skb)->nr_frags; i++) {
1017                 int end;
1018
1019                 BUG_TRAP(start <= offset + len);
1020
1021                 end = start + skb_shinfo(skb)->frags[i].size;
1022                 if ((copy = end - offset) > 0) {
1023                         unsigned int csum2;
1024                         u8 *vaddr;
1025                         skb_frag_t *frag = &skb_shinfo(skb)->frags[i];
1026
1027                         if (copy > len)
1028                                 copy = len;
1029                         vaddr = kmap_skb_frag(frag);
1030                         csum2 = csum_partial(vaddr + frag->page_offset +
1031                                              offset - start, copy, 0);
1032                         kunmap_skb_frag(vaddr);
```

```
1033                        csum = csum_block_add(csum, csum2, pos);
1034                        if (!(len -= copy))
1035                                return csum;
1036                        offset += copy;
1037                        pos    += copy;
1038                }
1039                start = end;
1040        }
1041
1042        if (skb_shinfo(skb)->frag_list) {
1043                struct sk_buff *list = skb_shinfo(skb)->frag_list;
1044
1045                for (; list; list = list->next) {
1046                        int end;
1047
1048                        BUG_TRAP(start <= offset + len);
1049
1050                        end = start + list->len;
1051                        if ((copy = end - offset) > 0) {
1052                                unsigned int csum2;
1053                                if (copy > len)
1054                                        copy = len;
1055                                csum2 = skb_checksum(list, offset - start,
1056                                                     copy, 0);
1057                                csum = csum_block_add(csum, csum2, pos);
1058                                if ((len -= copy) == 0)
1059                                        return csum;
1060                                offset += copy;
1061                                pos    += copy;
1062                        }
1063                        start = end;
1064                }
1065        }
1066        if (len)
1067                BUG();
1068
1069        return csum;
1070 }
```

Autrement dit on calcule la somme de contrôle partielle de l'en-tête à laquelle on ajoute les sommes de contrôle partielles des fragments du tableau des fragments puis celles des tampons de la liste des fragments.

Les fonctions auxiliaires `csum_add()` et `csum_block_add()` sont définies dans le fichier `linux/include/net/checksum.h`:

Code Linux 2.6.10

```
60 static inline unsigned int csum_add(unsigned int csum, unsigned int addend)
61 {
62         csum += addend;
63         return csum + (csum < addend);
64 }
[...]
71 static inline unsigned int
72 csum_block_add(unsigned int csum, unsigned int csum2, int offset)
73 {
74         if (offset&1)
75                 csum2 = ((csum2&0xFF00FF)<<8)+((csum2>>8)&0xFF00FF);
76         return csum_add(csum, csum2);
77 }
```

25.2 Réception des datagrammes UDP

Nous avons vu au chapitre 23 que le paquet reçu par la couche IP est transmis à la couche transport, en traitant le descripteur de tampon grâce à la méthode `handler()` de l'instance de structure `inet_protocol` associée au type de communication. Nous avons vu aussi que dans le cas d'UDP, cette instance s'appelle `udp_protocol` et le nom de la fonction `udp_rcv()`.

25.2.1 Fonction de traitement principale

Code Linux 2.6.10

La fonction `udp_rcv()` est définie dans le fichier `linux/net/ipv4/udp.c`:

```
1110 /*
1111  *      Tout ce que nous avons besoin de faire est d'obtenir la socket, et alors
              d'effectuer une verification de la somme de controle.
1112  */
1113
1114 int udp_rcv(struct sk_buff *skb)
1115 {
1116         struct sock *sk;
1117         struct udphdr *uh;
1118         unsigned short ulen;
1119         struct rtable *rt = (struct rtable*)skb->dst;
1120         u32 saddr = skb->nh.iph->saddr;
1121         u32 daddr = skb->nh.iph->daddr;
1122         int len = skb->len;
1123
1124         /*
1125          *      Valider le paquet et la longueur UDP.
1126          */
1127         if (!pskb_may_pull(skb, sizeof(struct udphdr)))
1128                 goto no_header;
1129
1130         uh = skb->h.uh;
1131
1132         ulen = ntohs(uh->len);
1133
1134         if (ulen > len || ulen < sizeof(*uh))
1135                 goto short_packet;
1136
1137         if (pskb_trim(skb, ulen))
1138                 goto short_packet;
1139
1140         if (udp_checksum_init(skb, uh, ulen, saddr, daddr) < 0)
1141                 goto csum_error;
1142
1143         if(rt->rt_flags & (RTCF_BROADCAST|RTCF_MULTICAST))
1144                 return udp_v4_mcast_deliver(skb, uh, saddr, daddr);
1145
1146         sk = udp_v4_lookup(saddr, uh->source, daddr, uh->dest, skb->dev->ifindex);
1147
1148         if (sk != NULL) {
1149                 int ret = udp_queue_rcv_skb(sk, skb);
1150                 sock_put(sk);
1151
1152                 /* une valeur de retour > 0 signifie de soumettre a nouveau l'entree,
1153                  * mais il faut que le retour soit -protocol ou 0
1154                  */
1155                 if (ret > 0)
1156                         return -ret;
1157                 return 0;
1158         }
```

```
1159
1160         if (!xfrm4_policy_check(NULL, XFRM_POLICY_IN, skb))
1161                 goto drop;
1162
1163         /* Pas de socket. Ecarter le paquet sans faire de bruit si la somme de controle
                est fausse */
1164         if (udp_checksum_complete(skb))
1165                 goto csum_error;
1166
1167         UDP_INC_STATS_BH(UDP_MIB_NOPORTS);
1168         icmp_send(skb, ICMP_DEST_UNREACH, ICMP_PORT_UNREACH, 0);
1169
1170         /*
1171          * Hum. Nous obtenons un paquet UDP pour un port qui n'est pas a
1172          * l'ecoute. Ignorons-le.
1173          */
1174         kfree_skb(skb);
1175         return(0);
1176
1177 short_packet:
1178         NETDEBUG(if (net_ratelimit())
1179                 printk(KERN_DEBUG "UDP: short packet: From %u.%u.%u.%u:%u %d/%d
                                                          to %u.%u.%u.%u:%u\n",
1180                         NIPQUAD(saddr),
1181                         ntohs(uh->source),
1182                         ulen,
1183                         len,
1184                         NIPQUAD(daddr),
1185                         ntohs(uh->dest)));
1186 no_header:
1187         UDP_INC_STATS_BH(UDP_MIB_INERRORS);
1188         kfree_skb(skb);
1189         return(0);
1190
1191 csum_error:
1192         /*
1193          * RFC1122 : OK. Ecarter le mauvais paquet sans faire de bruit (pour autant que
1194          * le reseau est concerne) comme pour 4.1.3.4 (MUST).
1195          */
1196         NETDEBUG(if (net_ratelimit())
1197                  printk(KERN_DEBUG "UDP: bad checksum. From %d.%d.%d.%d:%d
                                                           to %d.%d.%d.%d:%d ulen %d\n",
1198                         NIPQUAD(saddr),
1199                         ntohs(uh->source),
1200                         NIPQUAD(daddr),
1201                         ntohs(uh->dest),
1202                         ulen));
1203 drop:
1204         UDP_INC_STATS_BH(UDP_MIB_INERRORS);
1205         kfree_skb(skb);
1206         return(0);
1207 }
```

Autrement dit :

- On déclare une adresse de descripteur de couche transport, un en-tête UDP et une longueur de datagramme UDP.
- On déclare une entrée de cache de routage, que l'on instantie avec celle associée au descripteur de tampon de socket passé en argument.

  Nous avons vu au chapitre 23 comment ce champ entrée de cache de routage a été renseigné.

- On déclare une adresse IP source, que l'on instantie avec celle associée à l'en-tête IP du descripteur de tampon passé en argument.
- On déclare une adresse IP destination, que l'on instantie avec celle associée à l'en-tête IP du descripteur de tampon passé en argument.
- On déclare une longueur, que l'on instantie avec celle associée au descripteur de tampon passé en argument.
- On vérifie la longueur du datagramme :
  - Si la longueur du datagramme est inférieure à la taille minimum d'un en-tête UDP, on incrémente le nombre d'erreurs sur les datagrammes UDP entrants, on libère le descripteur de tampon et on renvoie 0.

    Les informations statistiques sur UDP sont détenues dans une variable du type `udp_mib`. Celui-ci est défini dans le fichier en-tête `linux/include/net/snmp.h` :

Code Linux 2.6.10

```
97  /* UDP */
98  #define UDP_MIB_MAX     __UDP_MIB_MAX
99  struct udp_mib {
100         unsigned long   mibs[UDP_MIB_MAX];
101 } __SNMP_MIB_ALIGN__;
```

    La macro `UDP_INC_STATS_BH()` est définie dans le fichier en-tête `linux/include/net/udp.h` :

Code Linux 2.6.10

```
77 DECLARE_SNMP_STAT(struct udp_mib, udp_statistics);
78 #define UDP_INC_STATS(field)            SNMP_INC_STATS(udp_statistics, field)
79 #define UDP_INC_STATS_BH(field)         SNMP_INC_STATS_BH(udp_statistics, field)
80 #define UDP_INC_STATS_USER(field)       SNMP_INC_STATS_USER(udp_statistics, field)
```

  - On instantie l'adresse d'en-tête UDP avec celle du champ correspondant du descripteur de tampon passé en argument.
  - On initialise la longueur de datagramme UDP avec celle indiquée dans l'en-tête UDP.
  - Si cette longueur de datagramme UDP est plus grande que la longueur du datagramme décrit par le descripteur de tampon passé en argument ou est plus petite que la taille minimale d'un en-tête UDP, on affiche un message noyau, on incrémente le nombre d'erreurs sur les datagrammes UDP entrants, on libère le descripteur de tampon et on renvoie 0.
- On vérifie la somme de contrôle, grâce à la fonction `udp_checksum_init()` étudiée ci-après. Si elle ne concorde pas avec celle indiquée dans l'en-tête UDP, on affiche un message noyau, on incrémente le nombre d'erreurs sur les datagrammes UDP entrants, on libère le descripteur de tampon et on renvoie 0.
- Les datagrammes de multidiffusion et de diffusion générale sont transmis à la fonction `udp_v4_mcast_deliver()`, qui ne nous intéressera pas ici.
- On essaie d'instantier le descripteur de couche transport grâce à la fonction `udp_v4_lookup()` de consultation de la table `udp_hash`, qui sera étudiée ci-dessous.
- Si on y parvient :
  - On place ce descripteur dans la file d'attente UDP en réception dans laquelle il sera récupéré par l'utilisateur, grâce à la fonction `udp_queue_rcv_skb()` qui sera étudiée ci-dessous.
  - On décrémente le compteur de référence de ce descripteur de couche de transport.
  - Si une erreur est intervenue, on renvoie l'opposé du code d'erreur fourni par la fonction précédente. Sinon on renvoie 0.

- Si on n'est pas parvenu à instantier le descripteur de couche transport :
  - Si c'est dû à un problème de politique, on détruit le datagramme : on incrémente le nombre d'erreurs sur les datagrammes UDP entrants, on libère le descripteur de tampon et on renvoie 0.
  - Si ceci est dû à un problème de somme de contrôle, vérifié grâce à la fonction `udp_checksum_complete()`, on affiche un message noyau, on incrémente le nombre d'erreurs sur les datagrammes UDP entrants, on libère le descripteur de tampon et on renvoie 0.
  - Dans les autres cas, c'est que le port n'est pas accessible : on incrémente le nombre d'erreurs sur les datagrammes UDP entrants pour lesquels le port n'est pas accessible, on envoie un message ICMP, on libère le descripteur de tampon et on renvoie 0.

25.2.2 Vérification rapide de la somme de contrôle

La fonction `udp_checksum_init()` est définie dans le fichier `linux/net/ipv4/udp.c` : Code Linux 2.6.10

```
1085 /* Initialise la verification de la somme de controle UDP. Si on la quitte avec
        la valeur zero (succes),
1086  * CHECKSUM_UNNECESSARY signifie qu'on n'a pas besoin d'autres verifications.
1087  * Sinon la verification complete necessite le corps du paquet,
1088  * incluant l'en-tete udp et l'incorporant dans skb->csum.
1089  */
1090 static int udp_checksum_init(struct sk_buff *skb, struct udphdr *uh,
1091                              unsigned short ulen, u32 saddr, u32 daddr)
1092 {
1093         if (uh->check == 0) {
1094                 skb->ip_summed = CHECKSUM_UNNECESSARY;
1095         } else if (skb->ip_summed == CHECKSUM_HW) {
1096                 skb->ip_summed = CHECKSUM_UNNECESSARY;
1097                 if (!udp_check(uh, ulen, saddr, daddr, skb->csum))
1098                         return 0;
1099                 NETDEBUG(if (net_ratelimit()) printk(KERN_DEBUG "udp v4 hw csum
                                                        failure.\n"));
1100                 skb->ip_summed = CHECKSUM_NONE;
1101         }
1102         if (skb->ip_summed != CHECKSUM_UNNECESSARY)
1103                 skb->csum = csum_tcpudp_nofold(saddr, daddr, ulen, IPPROTO_UDP, 0);
1104         /* Nous devrions probablement verifier l'en-tete udp (il devrait etre dans le cache
1105          * en tout cas) et les donnees dans les petits paquets (< rx copybreak).
1106          */
1107         return 0;
1108 }
```

Autrement dit :

- Si la valeur de la somme de contrôle de l'en-tête UDP est nulle, c'est que l'émetteur n'a pas envoyé de somme de contrôle. On positionne alors le champ `ip_summed` du descripteur de tampon passé en argument à la valeur `CHECKSUM_UNNECESSARY`.
- Si la valeur de la somme de contrôle de l'en-tête UDP est non nulle et que la valeur du champ `ip_summed` du descripteur de tampon passé en argument est `CHECKSUM_HW` :
  - On positionne également le champ `ip_summed` du descripteur de tampon passé en argument à la valeur `CHECKSUM_UNNECESSARY`.
  - On vérifie qu'il s'agit de la bonne somme de contrôle grâce à la fonction `udp_check()` et on renvoie 0 en cas de succès.

Code Linux 2.6.10

La fonction `udp_check()` est définie dans le fichier `linux/net/ipv4/udp.c`:

```
475 static unsigned short udp_check(struct udphdr *uh, int len, unsigned long saddr,
                                    unsigned long daddr, unsigned long base)
476 {
477         return(csum_tcpudp_magic(saddr, daddr, len, IPPROTO_UDP, base));
478 }
```

qui renvoie à la fonction `csum_tcpudp_magic()`, qui dépend du microprocesseur. Celle-ci est définie dans le fichier `linux/include/asm-i386/checksum.h` pour les microprocesseurs Intel :

Code Linux 2.6.10

```
93  /*
94   *      Plie une somme de controle partielle
95   */
96
97  static inline unsigned int csum_fold(unsigned int sum)
98  {
99          __asm__(
100                 "addl %1, %0            ;\n"
101                 "adcl $0xffff, %0       ;\n"
102                 : "=r" (sum)
103                 : "r" (sum << 16), "" (sum & 0xffff0000)
104         );
105         return (~sum) >> 16;
106 }
107
108 static inline unsigned long csum_tcpudp_nofold(unsigned long saddr,
109                                                unsigned long daddr,
110                                                unsigned short len,
111                                                unsigned short proto,
112                                                unsigned int sum)
113 {
114     __asm__(
115         "addl %1, %0    ;\n"
116         "adcl %2, %0    ;\n"
117         "adcl %3, %0    ;\n"
118         "adcl $0, %0    ;\n"
119         : "=r" (sum)
120         : "g" (daddr), "g"(saddr), "g"((ntohs(len)<<16)+proto*256), ""(sum));
121     return sum;
122 }
123
124 /*
125  * calcule la somme de controle du pseudo en-tete TCP/UDP
126  * renvoie une somme de controle sur 16 bits, deja complementee
127  */
128 static inline unsigned short int csum_tcpudp_magic(unsigned long saddr,
129                                                unsigned long daddr,
130                                                unsigned short len,
131                                                unsigned short proto,
132                                                unsigned int sum)
133 {
134         return csum_fold(csum_tcpudp_nofold(saddr,daddr,len,proto,sum));
135 }
```

  - Sinon on affiche un message noyau annonçant l'échec de la vérification et on positionne le champ `ip_summed` du descripteur de tampon passé en argument à la valeur `CHECKSUM_NONE`.

- Si la valeur du champ `ip_summed` du descripteur de tampon passé en argument est `CHECKSUM_UNNECESSARY`, on calcule la somme de contrôle que l'on place dans le champ `csum` du descripteur de tampon passé en argument et on renvoie 0.

25.2.3 Consultation de la table de hachage UDP

25.2.3.1 Appel de la consultation

La fonction en ligne `udp_v4_lookup()` est définie dans le fichier `linux/net/ipv4/udp.c`: Code Linux 2.6.10

```
266 __inline__ struct sock *udp_v4_lookup(u32 saddr, u16 sport, u32 daddr, u16 dport, int dif)
267 {
268         struct sock *sk;
269
270         read_lock(&udp_hash_lock);
271         sk = udp_v4_lookup_longway(saddr, sport, daddr, dport, dif);
272         if (sk)
273                 sock_hold(sk);
274         read_unlock(&udp_hash_lock);
275         return sk;
276 }
```

Autrement dit :

- On déclare un descripteur de couche transport.
- On verrouille en lecture la table de hachage UDP.
  Le verrou `udp_hash_lock` est défini plus haut dans le même fichier : Code Linux 2.6.10

  ```
  118 rwlock_t udp_hash_lock = RW_LOCK_UNLOCKED;
  ```

- On essaie d'instantier le descripteur de couche transport grâce à la fonction `udp_v4_lookup_longway()` étudiée ci-dessous.
- Si on y parvient, on incrémente son compteur de référence.
- On déverrouille la lecture dans la table de hachage UDP et on renvoie l'adresse du descripteur (`NULL` si on n'est pas parvenue à l'instantier).

25.2.3.2 Options Internet

Nous allons avoir besoin d'une structure de données supplémentaires. La structure `inet_opt` est définie dans le fichier `linux/include/linux/ip.h`: Code Linux 2.6.10

```
110 struct inet_opt {
111         /* Comparaisons de demultiplexage de socket sur les paquets entrants. */
112         __u32                   daddr;          /* Adresse IPv4 etrangere */
113         __u32                   rcv_saddr;      /* Adresse IPv4 liee localement */
114         __u16                   dport;          /* Port de destination */
115         __u16                   num;            /* Port local */
116         __u32                   saddr;          /* Source d'envoi */
117         int                     uc_ttl;         /* Unicast TTL */
118         int                     tos;            /* TOS */
119         unsigned                cmsg_flags;
120         struct ip_options       *opt;
121         __u16                   sport;          /* Port source */
122         unsigned char           hdrincl;        /* En-tetes incluses ? */
123         __u8                    mc_ttl;         /* Multicasting TTL */
124         __u8                    mc_loop;        /* Loopback */
125         __u8                    pmtudisc;
126         __u16                   id;             /* Compteur d'IDentification pour les
                                                       paquets DF */
127         unsigned                recverr : 1,
128                                 freebind : 1;
129         int                     mc_index;       /* Multicast device index */
130         __u32                   mc_addr;
131         struct ip_mc_socklist   *mc_list;       /* Tableau de groupe */
```

```
132         /*
133          * Les membres suivants sont utilises pour detenir les informations necessaires
134          * a la construction d'un en-tete ip pour chaque fragment ip lorsque la socket
               a un bouchon.
135          */
136         struct {
137                 unsigned int            flags;
138                 unsigned int            fragsize;
139                 struct ip_options       *opt;
140                 struct rtable           *rt;
141                 int                     length; /* Longueur totale de toutes les trames */
142                 u32                     addr;
143                 struct flowi            fl;
144         } cork;
145 };
```

25.2.3.3 Consultation proprement dite

Code Linux 2.6.10

La fonction `udp_v4_lookup_longway()` est définie dans le fichier `linux/net/ipv4/udp.c`:

```
219 /* UDP est presque toujours en dehors du wazoo, il est inutile d'essayer d'etre
220  * plus dur que ca. -DaveM
221  */
222 struct sock *udp_v4_lookup_longway(u32 saddr, u16 sport, u32 daddr, u16 dport, int dif)
223 {
224         struct sock *sk, *result = NULL;
225         struct hlist_node *node;
226         unsigned short hnum = ntohs(dport);
227         int badness = -1;
228
229         sk_for_each(sk, node, &udp_hash[hnum & (UDP_HTABLE_SIZE - 1)]) {
230                 struct inet_opt *inet = inet_sk(sk);
231
232                 if (inet->num == hnum && !ipv6_only_sock(sk)) {
233                         int score = (sk->sk_family == PF_INET ? 1 : 0);
234                         if (inet->rcv_saddr) {
235                                 if (inet->rcv_saddr != daddr)
236                                         continue;
237                                 score+=2;
238                         }
239                         if (inet->daddr) {
240                                 if (inet->daddr != saddr)
241                                         continue;
242                                 score+=2;
243                         }
244                         if (inet->dport) {
245                                 if (inet->dport != sport)
246                                         continue;
247                                 score+=2;
248                         }
249                         if (sk->sk_bound_dev_if) {
250                                 if (sk->sk_bound_dev_if != dif)
251                                         continue;
252                                 score+=2;
253                         }
254                         if(score == 9) {
255                                 result = sk;
256                                 break;
257                         } else if(score > badness) {
258                                 result = sk;
259                                 badness = score;
260                         }
```

```
261                 }
262         }
263         return result;
264 }
```

Autrement dit :

- On déclare deux descripteurs de couche transport : un servant d'index dans la table de hachage UDP et l'autre pour recueillir le résultat, ce dernier étant initialisé à `NULL`.
- On déclare une liste de hachage de nœuds.
- On déclare un numéro de hachage, que l'on initialise avec le numéro de port de destination passé en argument.
- On déclare un entier de test, que l'on initialise à -1.
- On parcourt la table de hachage UDP.

  La table `udp_hash[]` de hachage UDP est définie dans le fichier `linux/net/ipv4/udp.c` : Code Linux 2.6.10

```
117 struct hlist_head udp_hash[UDP_HTABLE_SIZE];
```

  La taille `UDP_HTABLE_SIZE` de la table de hachage UDP est définie dans le fichier `linux/include/net/udp.h` : Code Linux 2.6.10

```
32 #define UDP_HTABLE_SIZE         128
```

  La macro `sk_for_each()` est définie dans le fichier `linux/include/net/sock.h` : Code Linux 2.6.10

```
371 #define sk_for_each(__sk, node, list) \
372         hlist_for_each_entry(__sk, node, list, sk_node)
```

- Pour chaque élément, on déclare une option Internet, que l'on instantie avec le champ correspondant du descripteur de couche transport en train d'être analysé.

  La fonction en ligne `inet_sk()` est définie dans le fichier `linux/include/linux/ip.h` : Code Linux 2.6.10

```
151 /* ATTENTION : ne pas changer la place des  membres de inet_sock ! */
152 struct inet_sock {
153         struct sock       sk;
154 #if defined(CONFIG_IPV6) || defined(CONFIG_IPV6_MODULE)
155         struct ipv6_pinfo *pinet6;
156 #endif
157         struct inet_opt   inet;
158 };
159
160 static inline struct inet_opt * inet_sk(const struct sock *__sk)
161 {
162         return &((struct inet_sock *)__sk)->inet;
163 }
```

  La macro `ipv6_only_sock()`, qui ne nous concerne pas vraiment ici, est définie dans le fichier `linux/include/linux/ipv6.h` : Code Linux 2.6.10

```
292 #if defined(CONFIG_IPV6) || defined(CONFIG_IPV6_MODULE)
293 #define __ipv6_only_sock(sk)    (inet6_sk(sk)->ipv6only)
294 #define ipv6_only_sock(sk)      ((sk)->sk_family == PF_INET6 && __ipv6_only_sock(sk))
295 #else
296 #define __ipv6_only_sock(sk)    0
297 #define ipv6_only_sock(sk)      0
298 #endif
```

- Si le champ numéro de l'option Internet est égal au numéro de hachage et s'il ne s'agit pas d'un descripteur de couche de transport spécifique à IPv6 :
  - On déclare un score, que l'on initialise à 1 ou à 0 suivant que la famille de protocoles indiquée dans le descripteur de couche de transport est IPv4 ou non.
  - Si l'adresse source reçue est définie dans l'option Internet : si elle est différente de l'adresse de destination passée en argument, on sort de la boucle ; sinon on augmente le score de 2.
  - Si l'adresse de destination est définie dans l'option Internet : si elle est différente de l'adresse source passée en argument, on sort de la boucle ; sinon on augmente le score de 2.
  - Si le numéro de port de destination est défini dans l'option Internet : s'il est différent du numéro de port source passé en argument, on sort de la boucle ; sinon on augmente le score de 2.
  - Si l'index d'interface réseau est défini dans le descripteur de transport en train d'être analysé : s'il est différent de l'index d'interface de destination passé en argument, on sort de la boucle ; sinon on augmente le score de 2.
  - Si le score est égal à 9, on a trouvé le descripteur de couche de transport qu'il fallait. On en renvoie l'adresse.
  - Sinon, si le score est supérieur à la valeur de test, le résultat prend (provisoirement) la valeur du descripteur de couche de transport en train d'être analysé et la valeur de test ce score. Ceci permettra de renvoyer le descripteur de couche de transport qui correspond au mieux aux critères.
- Si on n'a pas trouvé le descripteur de couche de transport qui répond à tous les critères, on renvoie celui qui en remplit le plus.

25.2.4 Traitement du datagramme UDP

Code Linux 2.6.10

La fonction `udp_queue_rcv_skb()` est définie dans le fichier `linux/net/ipv4/udp.c` :

```
978  /* renvoie :
979   *  -1 : erreur
980   *   0 : succes
981   *  >0 : on soumet a nouveau le protocole "udp encap"
982   *
983   * Noter qu'en cas de succes ou d'erreur, le skb est suppose avoir
984   * ete place dans la file d'attente ou libere.
985   */
986  static int udp_queue_rcv_skb(struct sock * sk, struct sk_buff *skb)
987  {
988          struct udp_opt *up = udp_sk(sk);
989
990          /*
991           *      Charger-le a la socket, ecarter-le si la file d'attente est pleine.
992           */
993          if (!xfrm4_policy_check(sk, XFRM_POLICY_IN, skb)) {
994                  kfree_skb(skb);
995                  return -1;
996          }
997
998          if (up->encap_type) {
999                  /*
1000                  * C'est une socket d'encapsulation, regardons donc si c'est
1001                  * un paquet encapsule.
```

```
1002                 * S'il s'agit d'un paquet a garder en vie, contentons-nous de le manger.
1003                 * S'il s'agit d'un paquet encapsule, passons-le en entree
1004                 * xfrm IPsec et renvoyons la reponse
1005                 * appropriee. Sinon, passons a travers et
1006                 * passons-le a la socket UDP.
1007                 */
1008                int ret;
1009
1010                ret = udp_encap_rcv(sk, skb);
1011                if (ret == 0) {
1012                        /* Mangeons le paquet ... */
1013                        kfree_skb(skb);
1014                        return 0;
1015                }
1016                if (ret < 0) {
1017                        /* traitons le paquet ESP */
1018                        ret = xfrm4_rcv_encap(skb, up->encap_type);
1019                        UDP_INC_STATS_BH(UDP_MIB_INDATAGRAMS);
1020                        return -ret;
1021                }
1022                /* PAS PASSE A TRAVERS -- c'est un paquet UDP */
1023        }
1024
1025        if (sk->sk_filter && skb->ip_summed != CHECKSUM_UNNECESSARY) {
1026                if (__udp_checksum_complete(skb)) {
1027                        UDP_INC_STATS_BH(UDP_MIB_INERRORS);
1028                        kfree_skb(skb);
1029                        return -1;
1030                }
1031                skb->ip_summed = CHECKSUM_UNNECESSARY;
1032        }
1033
1034        if (sock_queue_rcv_skb(sk,skb)<0) {
1035                UDP_INC_STATS_BH(UDP_MIB_INERRORS);
1036                kfree_skb(skb);
1037                return -1;
1038        }
1039        UDP_INC_STATS_BH(UDP_MIB_INDATAGRAMS);
1040        return 0;
1041 }
```

Autrement dit :

– On déclare une adresse d'option UDP, que l'on instantie avec l'adresse du champ correspondant du descripteur de couche transport passé en argument.

La structure `udp_opt` est définie dans le fichier `linux/include/linux/udp.h`: Code Linux 2.6.10

```
43 struct udp_opt {
44         int             pending;        /* Y a-t-il des trames en attente ?      */
45         unsigned int    corkflag;       /* Un bouchon est necessaire             */
46         __u16           encap_type;     /* Est-ce une socket d'encapsulation ?   */
47         /*
48          * Le membre suivant contient les infomations pour creer un en-tete UDP
49          * lorsque la socket a un bouchon.
50          */
51         __u16           len;            /* longueur totale des trames en attente */
52 };
```

La fonction en ligne `udp_sk()` est définie dans le fichier `linux/include/linux/udp.h`: Code Linux 2.6.10

```
54 /* ATTENTION ! ne pas changer la disposition des membres dans udp_sock */
55 struct udp_sock {
```

```
56          struct sock       sk;
57 #if defined(CONFIG_IPV6) || defined(CONFIG_IPV6_MODULE)
58          struct ipv6_pinfo *pinet6;
59 #endif
60          struct inet_opt   inet;
61          struct udp_opt    udp;
62 };
63
64 static inline struct udp_opt * udp_sk(const struct sock *__sk)
65 {
66          return &((struct udp_sock *)__sk)->udp;
67 }
```

- Si la file d'attente est pleine, on libère le descripteur de tampon de socket passé en argument et on renvoie -1.
- S'il s'agit d'une socket d'encapsulation, on la traite comme indiqué dans le commentaire, ce qui ne nous intéresse pas vraiment dans cet ouvrage.
- S'il s'agit d'un filtre et si le champ `ip_summed` du descripteur de tampon passé en argument ne spécifie pas qu'il n'est pas nécessaire de vérifier la somme de contrôle :

  - On calcule la somme de contrôle. Si elle ne concorde pas, on incrémente l'information statistique sur le nombre d'erreurs des datagrammes UDP entrants, on libère le descripteur de tampon passé en argument et on renvoie -1.

    La fonction en ligne `__udp_checksum_complete()` est définie dans le fichier `linux/net/ipv4/udp.c` :

Code Linux 2.6.10

```
757 static __inline__ int __udp_checksum_complete(struct sk_buff *skb)
758 {
759          return (unsigned short)csum_fold(skb_checksum(skb, 0, skb->len,
                         skb->csum));
760 }
```

  - Sinon on modifie le champ `ip_summed` du descripteur de tampon passé en argument en spécifiant maintenant qu'il n'est pas nécessaire de vérifier la somme de contrôle.

- Sinon on essaie de placer le datagramme dans la file d'attente des datagrammes reçus grâce à la fonction `sock_queue_rcv_skb()`.

  C'est le cas qui nous intéresse.
- Si on n'y parvient pas, on incrémente l'information statistique sur le nombre d'erreurs des datagrammes UDP entrants, on libère le descripteur de tampon passé en argument et on renvoie -1.
- Sinon on incrémente l'information statistique sur le nombre de datagrammes UDP reçus avec succès, on libère le descripteur de tampon passé en argument et on renvoie 0.

25.2.5 Mise dans la file d'attente de réception UDP

La fonction en ligne `sock_queue_rcv_skb()` est définie dans le fichier `linux/include/net/sock.h` :

Code Linux 2.6.10

```
1070 static inline int sock_queue_rcv_skb(struct sock *sk, struct sk_buff *skb)
1071 {
1072         int err = 0;
1073         int skb_len;
1074
1075         /* Convertit skb->rcvbuf en unsigned... C'est sans importance mais ca reduit
1076            le nombre d'avertissements lorsqu'on compile avec -W --ANK
```

```
1077          */
1078         if (atomic_read(&sk->sk_rmem_alloc) + skb->truesize >=
1079             (unsigned)sk->sk_rcvbuf) {
1080                 err = -ENOMEM;
1081                 goto out;
1082         }
1083
1084         /* Ceci devrait etre un blocage si sock_queue_rcv_skb est utilise
1085            avec un verrou de socket ! Nous supposons que les utilisateurs de cette
1086            fonction n'utilisent pas de verrou.
1087         */
1088         err = sk_filter(sk, skb, 1);
1089         if (err)
1090                 goto out;
1091
1092         skb->dev = NULL;
1093         skb_set_owner_r(skb, sk);
1094
1095         /* Mise en cache de la longueur du SKB avant que nous mettions dans la file
1096          * d'attente en reception. Une fois qu'il a ete ajoute, il ne nous appartient
1097          * plus et il peut etre libere par d'autres threads de controle
1098          * de depilement des paquets de la file d'attente.
1099          */
1100         skb_len = skb->len;
1101
1102         skb_queue_tail(&sk->sk_receive_queue, skb);
1103
1104         if (!sock_flag(sk, SOCK_DEAD))
1105                 sk->sk_data_ready(sk, skb_len);
1106 out:
1107         return err;
1108 }
```