Chapitre 33

Envoi de paquets ordinaires sous IPv4

Nous avons vu au chapitre 32 que la fonction UDP d'envoi de datagramme fait appel aux fonctions `ip_route_output_flow()` de redirection en sortie et `ip_push_pending_frames()` de constitution du paquet IP et d'envoi de celui-ci. Nous allons étudier ces fonctions dans ce chapitre.

33.1 Première étape : routage d'un paquet ordinaire

Comme dans le cas des paquets entrants, la redirection d'un paquet sortant se fait d'abord par recherche dans le cache de routage et, en cas d'échec dans celui-ci, directement par requête à la FIB.

33.1.1 Recherche dans le cache de routage

Code Linux 2.6.10

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

```
2214 int ip_route_output_flow(struct rtable **rp, struct flowi *flp, struct sock *sk, int flags)
2215 {
2216         int err;
2217
2218         if ((err = __ip_route_output_key(rp, flp)) != 0)
2219                 return err;
2220
2221         if (flp->proto) {
2222                 if (!flp->fl4_src)
2223                         flp->fl4_src = (*rp)->rt_src;
2224                 if (!flp->fl4_dst)
2225                         flp->fl4_dst = (*rp)->rt_dst;
2226                 return xfrm_lookup((struct dst_entry **)rp, flp, sk, flags);
2227         }
2228
2229         return 0;
2230 }
```

Elle fait essentiellement appel à la fonction `__ip_route_output_key()`.

La fonction `__ip_route_output_key()` est chargée de fournir une entrée de cache de routage à partir du flux Internet générique passé en paramètre. La procédure correspond tout d'abord à celle utilisée dans la fonction `ip_route_input()` : en d'autres termes, la fonction recherche dans le cache de routage s'il existe une entrée correspondante. Si aucune entrée n'a pu être trouvée, la procédure passe la main à `ip_route_output_slow()` pour effectuer une requête FIB.

Code Linux 2.6.10

Elle est définie dans le fichier `linux/net/ipv4/route.c` :

```
2180 int __ip_route_output_key(struct rtable **rp, const struct flowi *flp)
2181 {
2182         unsigned hash;
2183         struct rtable *rth;
2184
2185         hash = rt_hash_code(flp->fl4_dst, flp->fl4_src ^ (flp->oif << 5), flp->fl4_tos);
2186
2187         rcu_read_lock_bh();
2188         for (rth = rcu_dereference(rt_hash_table[hash].chain); rth;
2189                 rth = rcu_dereference(rth->u.rt_next)) {
2190                 if (rth->fl.fl4_dst == flp->fl4_dst &&
2191                     rth->fl.fl4_src == flp->fl4_src &&
2192                     rth->fl.iif == 0 &&
2193                     rth->fl.oif == flp->oif &&
2194 #ifdef CONFIG_IP_ROUTE_FWMARK
2195                     rth->fl.fl4_fwmark == flp->fl4_fwmark &&
2196 #endif
2197                     !((rth->fl.fl4_tos ^ flp->fl4_tos) &
2198                             (IPTOS_RT_MASK | RTO_ONLINK))) {
2199                         rth->u.dst.lastuse = jiffies;
2200                         dst_hold(&rth->u.dst);
2201                         rth->u.dst.__use++;
2202                         RT_CACHE_STAT_INC(out_hit);
```

```
2203                        rcu_read_unlock_bh();
2204                        *rp = rth;
2205                        return 0;
2206                }
2207                RT_CACHE_STAT_INC(out_hlist_search);
2208        }
2209        rcu_read_unlock_bh();
2210
2211        return ip_route_output_slow(rp, flp);
2212 }
```

Autrement dit :

- On déclare un index de hachage et une entrée de cache de routage.
- On calcule l'index de hachage à partir des clés fournies par le flux Internet passé en argument.
- On verrouille le dispositif d'écriture/mise à jour.
- On recherche dans la liste chaînée associée à l'index de hachage si on trouve une entrée de cache de routage dont les clès correspondent.

  La constante symbolique `RTO_ONLINK` est définie dans le fichier `linux/include/net/route.h` : Code Linux 2.6.10

  ```
  41 #define RTO_ONLINK      0x01
  ```

- Si on en trouve une :
  - On met à jour l'heure précise de dernière utilisation de cette entrée avec l'heure en cours.
  - On incrémente le nombre de références à cette entrée de cache de routage.
  - On met à jour une information statistique à ce propos.
  - On déverrouille le dispositif d'écriture/mise à jour.
  - L'argument adresse d'entrée de cache de routage prend comme valeur l'adresse de cette entrée et on renvoie 0.
- Si on n'en touve pas :
  - On met à jour une information statistique à ce propos.
  - On déverrouille le dispositif d'écriture/mise à jour.
  - On fait appel à la fonction `ip_route_output_slow()` pour consulter la FIB et on renvoie le code fourni par celle-ci.

33.1.2 Recherche étendue à la FIB

Lorsque le cache de routage ne comporte aucune entrée pour la destination d'un paquet IP créé localement, la fonction `ip_route_output_slow()` est appelée. Cette fonction soumet une requête FIB et un certain nombre de cas particuliers, pour lesquels la fonction `fib_lookup()` est insuffisante, sont traités. Si on trouve un résultat, celui-ci est consigné dans le cache de routage et on retourne la nouvelle entrée (à travers un argument).

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

```
1883 /*
1884  * Routine majeure de resolution de route.
1885  */
1886
```

```
1887 static int ip_route_output_slow(struct rtable **rp, const struct flowi *oldflp)
1888 {
1889         u32 tos = oldflp->fl4_tos & (IPTOS_RT_MASK | RTO_ONLINK);
1890         struct flowi fl = { .nl_u = { .ip4_u =
1891                                       { .daddr = oldflp->fl4_dst,
1892                                         .saddr = oldflp->fl4_src,
1893                                         .tos = tos & IPTOS_RT_MASK,
1894                                         .scope = ((tos & RTO_ONLINK) ?
1895                                                   RT_SCOPE_LINK :
1896                                                   RT_SCOPE_UNIVERSE),
1897 #ifdef CONFIG_IP_ROUTE_FWMARK
1898                                         .fwmark = oldflp->fl4_fwmark
1899 #endif
1900                                       } },
1901                             .iif = loopback_dev.ifindex,
1902                             .oif = oldflp->oif };
1903         struct fib_result res;
1904         unsigned flags = 0;
1905         struct rtable *rth;
1906         struct net_device *dev_out = NULL;
1907         struct in_device *in_dev = NULL;
1908         unsigned hash;
1909         int free_res = 0;
1910         int err;
1911
1912         res.fi          = NULL;
1913 #ifdef CONFIG_IP_MULTIPLE_TABLES
1914         res.r           = NULL;
1915 #endif
1916
1917         if (oldflp->fl4_src) {
1918                 err = -EINVAL;
1919                 if (MULTICAST(oldflp->fl4_src) ||
1920                     BADCLASS(oldflp->fl4_src) ||
1921                     ZERONET(oldflp->fl4_src))
1922                         goto out;
1923
1924                 /* C'est equivalent a inet_addr_type(saddr) == RTN_LOCAL */
1925                 dev_out = ip_dev_find(oldflp->fl4_src);
1926                 if (dev_out == NULL)
1927                         goto out;
1928
1929                 /* J'ai enleve la verification de == dev_out->oif ici.
1930                    C'etait faux pour deux raisons :
1931                    1. 1. ip_dev_find(saddr) peut renvoyer une fausse iface si saddr est
1932                       assignee a de multiples interfaces.
1933                    2. De plus, nous avons le droit d'envoyer des paquets avec saddr
1934                       d'une autre iface. --ANK
1935                  */
1936
1937                 if (oldflp->oif == 0
1938                     && (MULTICAST(oldflp->fl4_dst) || oldflp->fl4_dst == 0xFFFFFFFF)) {
1939                         /* Truc special : l'utilisateur peut diriger des multidiffusions
1940                            et des diffusions generales limitees via l'interface necessire
1941                            sans le chanter sur les toits avec IP_MULTICAST_IF ou IP_PKTINFO.
1942                            Ce truc ne sert pas qu'a s'amuser, il permet a
1943                            vic, vat et ses amis de marcher.
1944                            Ils lient les sockets a loopback, positionne ttl a zero
1945                            et s'attend a ce que ca marche.
1946                            Du point de vue du cache de routage ils ont disparus,
1947                            parce que nous n'avons pas le droit de construire un chemin de
1948                            multidiffusion avec une adr source loopback, le cache de
```

```
1949                            routage ne peut pas savoir que ttl est nul, donc ce paquet
1950                            ne quittera pas cet hote et la route est valide).
1951                            Heureusement ce truc est du bon travail.
1952                         */
1953
1954                        fl.oif = dev_out->ifindex;
1955                        goto make_route;
1956                }
1957                if (dev_out)
1958                        dev_put(dev_out);
1959                dev_out = NULL;
1960        }
1961        if (oldflp->oif) {
1962                dev_out = dev_get_by_index(oldflp->oif);
1963                err = -ENODEV;
1964                if (dev_out == NULL)
1965                        goto out;
1966                if (__in_dev_get(dev_out) == NULL) {
1967                        dev_put(dev_out);
1968                        goto out;       /* Faux code d'erreur */
1969                }
1970
1971                if (LOCAL_MCAST(oldflp->fl4_dst) || oldflp->fl4_dst == 0xFFFFFFFF) {
1972                        if (!fl.fl4_src)
1973                                fl.fl4_src = inet_select_addr(dev_out, 0,
1974                                                              RT_SCOPE_LINK);
1975                        goto make_route;
1976                }
1977                if (!fl.fl4_src) {
1978                        if (MULTICAST(oldflp->fl4_dst))
1979                                fl.fl4_src = inet_select_addr(dev_out, 0,
1980                                                              fl.fl4_scope);
1981                        else if (!oldflp->fl4_dst)
1982                                fl.fl4_src = inet_select_addr(dev_out, 0,
1983                                                              RT_SCOPE_HOST);
1984                }
1985        }
1986
1987        if (!fl.fl4_dst) {
1988                fl.fl4_dst = fl.fl4_src;
1989                if (!fl.fl4_dst)
1990                        fl.fl4_dst = fl.fl4_src = htonl(INADDR_LOOPBACK);
1991                if (dev_out)
1992                        dev_put(dev_out);
1993                dev_out = &loopback_dev;
1994                dev_hold(dev_out);
1995                fl.oif = loopback_dev.ifindex;
1996                res.type = RTN_LOCAL;
1997                flags |= RTCF_LOCAL;
1998                goto make_route;
1999        }
2000
2001        if (fib_lookup(&fl, &res)) {
2002                res.fi = NULL;
2003                if (oldflp->oif) {
2004                        /* Apparemment les tables de routage sont fausses. Supposons
2005                           que la destination est sur le lien.
2006
2007                           POURQUOI ? DW.
2008                           Parce que nous avons le droit d'envoyer a iface
2009                           meme s'il N'y a PAS de routes et PAS d'adresses
2010                           assignees. Lorsque oif est specifiee, les tables de routage
```

```
2011                            sont consultees dans un seul but :
2012                            voir si la destination passe par une passerelle plutot que
2013                            d'etre directe. De plus, si MSG_DONTROUTE est positionne,
2014                            nous envoyons le paquet, en ignorant et les tables de routages
2015                            et l'etat ifaddr. --ANK
2016
2017
2018                            Nous pourrions le faire meme si oif est inconnue,
2019                            comme dans IPv6, mais nous ne le faisons pas.
2020                          */
2021
2022                         if (fl.fl4_src == 0)
2023                                 fl.fl4_src = inet_select_addr(dev_out, 0,
2024                                                               RT_SCOPE_LINK);
2025                         res.type = RTN_UNICAST;
2026                         goto make_route;
2027                 }
2028                 if (dev_out)
2029                         dev_put(dev_out);
2030                 err = -ENETUNREACH;
2031                 goto out;
2032         }
2033         free_res = 1;
2034
2035         if (res.type == RTN_LOCAL) {
2036                 if (!fl.fl4_src)
2037                         fl.fl4_src = fl.fl4_dst;
2038                 if (dev_out)
2039                         dev_put(dev_out);
2040                 dev_out = &loopback_dev;
2041                 dev_hold(dev_out);
2042                 fl.oif = dev_out->ifindex;
2043                 if (res.fi)
2044                         fib_info_put(res.fi);
2045                 res.fi = NULL;
2046                 flags |= RTCF_LOCAL;
2047                 goto make_route;
2048         }
2049
2050 #ifdef CONFIG_IP_ROUTE_MULTIPATH
2051         if (res.fi->fib_nhs > 1 && fl.oif == 0)
2052                 fib_select_multipath(&fl, &res);
2053         else
2054 #endif
2055         if (!res.prefixlen && res.type == RTN_UNICAST && !fl.oif)
2056                 fib_select_default(&fl, &res);
2057
2058         if (!fl.fl4_src)
2059                 fl.fl4_src = FIB_RES_PREFSRC(res);
2060
2061         if (dev_out)
2062                 dev_put(dev_out);
2063         dev_out = FIB_RES_DEV(res);
2064         dev_hold(dev_out);
2065         fl.oif = dev_out->ifindex;
2066
2067 make_route:
2068         if (LOOPBACK(fl.fl4_src) && !(dev_out->flags&IFF_LOOPBACK))
2069                 goto e_inval;
2070
2071         if (fl.fl4_dst == 0xFFFFFFFF)
2072                 res.type = RTN_BROADCAST;
```

```
2073         else if (MULTICAST(fl.fl4_dst))
2074                 res.type = RTN_MULTICAST;
2075         else if (BADCLASS(fl.fl4_dst) || ZERONET(fl.fl4_dst))
2076                 goto e_inval;
2077
2078         if (dev_out->flags & IFF_LOOPBACK)
2079                 flags |= RTCF_LOCAL;
2080
2081         in_dev = in_dev_get(dev_out);
2082         if (!in_dev)
2083                 goto e_inval;
2084
2085         if (res.type == RTN_BROADCAST) {
2086                 flags |= RTCF_BROADCAST | RTCF_LOCAL;
2087                 if (res.fi) {
2088                         fib_info_put(res.fi);
2089                         res.fi = NULL;
2090                 }
2091         } else if (res.type == RTN_MULTICAST) {
2092                 flags |= RTCF_MULTICAST|RTCF_LOCAL;
2093                 if (!ip_check_mc(in_dev, oldflp->fl4_dst, oldflp->fl4_src, oldflp->proto))
2094                         flags &= ~RTCF_LOCAL;
2095                 /* Si la route de multidiffusion n'existe pas, utiliser
2096                    celle par defaut mais pas de passerelle dans ce cas.
2097                    Oui, c'est un truc.
2098                  */
2099                 if (res.fi && res.prefixlen < 4) {
2100                         fib_info_put(res.fi);
2101                         res.fi = NULL;
2102                 }
2103         }
2104
2105         rth = dst_alloc(&ipv4_dst_ops);
2106         if (!rth)
2107                 goto e_nobufs;
2108
2109         atomic_set(&rth->u.dst.__refcnt, 1);
2110         rth->u.dst.flags= DST_HOST;
2111         if (in_dev->cnf.no_xfrm)
2112                 rth->u.dst.flags |= DST_NOXFRM;
2113         if (in_dev->cnf.no_policy)
2114                 rth->u.dst.flags |= DST_NOPOLICY;
2115         rth->fl.fl4_dst = oldflp->fl4_dst;
2116         rth->fl.fl4_tos = tos;
2117         rth->fl.fl4_src = oldflp->fl4_src;
2118         rth->fl.oif     = oldflp->oif;
2119 #ifdef CONFIG_IP_ROUTE_FWMARK
2120         rth->fl.fl4_fwmark= oldflp->fl4_fwmark;
2121 #endif
2122         rth->rt_dst     = fl.fl4_dst;
2123         rth->rt_src     = fl.fl4_src;
2124         rth->rt_iif     = oldflp->oif ? : dev_out->ifindex;
2125         rth->u.dst.dev  = dev_out;
2126         dev_hold(dev_out);
2127         rth->idev       = in_dev_get(dev_out);
2128         rth->rt_gateway = fl.fl4_dst;
2129         rth->rt_spec_dst= fl.fl4_src;
2130
2131         rth->u.dst.output=ip_output;
2132
2133         RT_CACHE_STAT_INC(out_slow_tot);
2134
```

```
2135         if (flags & RTCF_LOCAL) {
2136                 rth->u.dst.input = ip_local_deliver;
2137                 rth->rt_spec_dst = fl.fl4_dst;
2138         }
2139         if (flags & (RTCF_BROADCAST | RTCF_MULTICAST)) {
2140                 rth->rt_spec_dst = fl.fl4_src;
2141                 if (flags & RTCF_LOCAL && !(dev_out->flags & IFF_LOOPBACK)) {
2142                         rth->u.dst.output = ip_mc_output;
2143                         RT_CACHE_STAT_INC(out_slow_mc);
2144                 }
2145 #ifdef CONFIG_IP_MROUTE
2146                 if (res.type == RTN_MULTICAST) {
2147                         if (IN_DEV_MFORWARD(in_dev) &&
2148                             !LOCAL_MCAST(oldflp->fl4_dst)) {
2149                                 rth->u.dst.input = ip_mr_input;
2150                                 rth->u.dst.output = ip_mc_output;
2151                         }
2152                 }
2153 #endif
2154         }
2155
2156         rt_set_nexthop(rth, &res, 0);
2157
2158
2159         rth->rt_flags = flags;
2160
2161         hash = rt_hash_code(oldflp->fl4_dst, oldflp->fl4_src ^ (oldflp->oif << 5), tos);
2162         err = rt_intern_hash(hash, rth, rp);
2163 done:
2164         if (free_res)
2165                 fib_res_put(&res);
2166         if (dev_out)
2167                 dev_put(dev_out);
2168         if (in_dev)
2169                 in_dev_put(in_dev);
2170 out:    return err;
2171
2172 e_inval:
2173         err = -EINVAL;
2174         goto done;
2175 e_nobufs:
2176         err = -ENOBUFS;
2177         goto done;
2178 }
```

Commentons le corps de cette fonction dans le cas d'un paquet ordinaire (ni multidiffusion ou diffusion générale ou autre). Autrement dit :

- Comme pour la fonction précédente le seul paramètre d'entrée est un flux Internet. Le résultat est une adresse d'entrée de cache de routage (apparaissant donc comme paramètre) et un code d'erreur (dont on espère qu'il sera nul pour indiquer qu'on a bien trouvé une entrée).
- On déclare un type de service, que l'on instantie à partir de celui fourni par le flux Internet passé en argument.
- On déclare un flux Internet, que l'on renseigne avec les champs du flux Internet passé en argument. Cependant on ne recopie pas les champs `iif` et `scope` : pour le premier, on suppose toujours que le périphérique *loopback* est utilisé ; pour le second, le champ est défini en fonction du drapeau `RTO_ONLINK` du champ `tos` du flux passé en argument et prend la valeur `RT_SCOPE_LINK` ou `RT_SCOPE_UNIVERSE`.

Ce flux Internet peut ensuite être modifié sans que les données passées en paramètre ne soient perdues.

- On déclare un résultat de consultation de la FIB, dont on renseigne le champ information `fi` à `NULL`.
- On examine, dans un premier temps, les paramètres d'entrée afin de déterminer s'ils comportent des erreurs ou s'ils correspondent à des cas justifiant un traitement particulier (lignes 1917–1999) :
  - Si l'adresse source est une adresse de multidiffusion, une mauvaise adresse ou une adresse nulle, on renvoie l'opposé du code d'erreur `EINVAL`.
  - Avec une adresse de destination de multidiffusion, on crée immédiatement une route sans effectuer de requête FIB lorsqu'une adresse source valide est spécifiée, à partir de laquelle un périphérique de sortie exploitable peut être identifié. Grâce à ce traitement spécifique, l'envoi des paquets de multidiffusion est simplifié.

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

    Code Linux 2.6.10

```
99  /*
100  *      Trouver le premier peripherique avec une adresse source donnee.
101  */
102
103 struct net_device * ip_dev_find(u32 addr)
104 {
105         struct flowi fl = { .nl_u = { .ip4_u = { .daddr = addr } } };
106         struct fib_result res;
107         struct net_device *dev = NULL;
108
109 #ifdef CONFIG_IP_MULTIPLE_TABLES
110         res.r = NULL;
111 #endif
112
113         if (!ip_fib_local_table ||
114             ip_fib_local_table->tb_lookup(ip_fib_local_table, &fl, &res))
115                 return NULL;
116         if (res.type != RTN_LOCAL)
117                 goto out;
118         dev = FIB_RES_DEV(res);
119
120         if (dev)
121                 dev_hold(dev);
122 out:
123         fib_res_put(&res);
124         return dev;
125 }
```

  - Si le flux Internet passé en argument spécifie une interface de sortie, on détermine une adresse source associée et, faute de spécifier une adresse de destination, on utilise l'adresse 127.0.0.1.

- Au terme de toutes ces opérations préparatoires, la requête FIB est exécutée (ligne 2001).

  Rappelons que celle-ci donnera toujours un résultat s'il existe une adresse par défaut.
- Si celle-ci ne donne aucun résultat, il est quand même possible de poursuivre dans certains cas : lorsqu'une interface de sortie est spécifiée dans le flux Internet passé en argument, celle-ci est utilisée, en supposant que la destination se trouve au sein du réseau auquel est connecté l'hôte, en spécifiant le type passerelle ou route directe.

– Le résultat de la requête permet de distinguer les adresses de destination locales de celles qui ne le sont pas. Dans le premier type d'adresses, on utilise toujours l'interface *loopback* tandis que dans le cas des adresses non locales, il peut être nécessaire d'utiliser la fonction `fib_select_multipath()` (ligne 2052) afin de choisir entre plusieurs routes, ou d'utiliser la fonction `fib_select_default()` (ligne 2056) pour choisir entre plusieurs routes par défaut.

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

Code Linux 2.6.10

```
166 static inline void fib_select_default(const struct flowi *flp, struct fib_result *res)
167 {
168         if (FIB_RES_GW(*res) && FIB_RES_NH(*res).nh_scope == RT_SCOPE_LINK)
169                 ip_fib_main_table->tb_select_default(ip_fib_main_table, flp, res);
170 }
```

– Les opérations s'achèvent avec le remplissage d'une nouvelle entrée de table de routage (à partir de la ligne 2105). Le pointeur `output()` prend comme valeur l'adresse de la fonction `ip_output()` (ligne 2131) et, dans le cas où l'adresse de destination se trouve sur le système local, le pointeur `input()` prend comme valeur l'adresse de la fonction `ip_local_deliver()` (ligne 2136).

– On fait appel à la fonction `rt_set_nexthop()` (ligne 2156), définie dans le fichier `linux/net/ipv4/route.c`:

Code Linux 2.6.10

```
1413 static void rt_set_nexthop(struct rtable *rt, struct fib_result *res, u32 itag)
1414 {
1415         struct fib_info *fi = res->fi;
1416
1417         if (fi) {
1418                 if (FIB_RES_GW(*res) &&
1419                     FIB_RES_NH(*res).nh_scope == RT_SCOPE_LINK)
1420                         rt->rt_gateway = FIB_RES_GW(*res);
1421                 memcpy(rt->u.dst.metrics, fi->fib_metrics,
1422                        sizeof(rt->u.dst.metrics));
1423                 if (fi->fib_mtu == 0) {
1424                         rt->u.dst.metrics[RTAX_MTU-1] = rt->u.dst.dev->mtu;
1425                         if (rt->u.dst.metrics[RTAX_LOCK-1] & (1 << RTAX_MTU) &&
1426                             rt->rt_gateway != rt->rt_dst &&
1427                             rt->u.dst.dev->mtu > 576)
1428                                 rt->u.dst.metrics[RTAX_MTU-1] = 576;
1429                 }
1430 #ifdef CONFIG_NET_CLS_ROUTE
1431                 rt->u.dst.tclassid = FIB_RES_NH(*res).nh_tclassid;
1432 #endif
1433         } else
1434                 rt->u.dst.metrics[RTAX_MTU-1]= rt->u.dst.dev->mtu;
1435
1436         if (rt->u.dst.metrics[RTAX_HOPLIMIT-1] == 0)
1437                 rt->u.dst.metrics[RTAX_HOPLIMIT-1] = sysctl_ip_default_ttl;
1438         if (rt->u.dst.metrics[RTAX_MTU-1] > IP_MAX_MTU)
1439                 rt->u.dst.metrics[RTAX_MTU-1] = IP_MAX_MTU;
1440         if (rt->u.dst.metrics[RTAX_ADVMSS-1] == 0)
1441                 rt->u.dst.metrics[RTAX_ADVMSS-1] =
                            max_t(unsigned int, rt->u.dst.dev->mtu - 40,
1442                                               ip_rt_min_advmss);
1443         if (rt->u.dst.metrics[RTAX_ADVMSS-1] > 65535 - 40)
1444                 rt->u.dst.metrics[RTAX_ADVMSS-1] = 65535 - 40;
1445
1446 #ifdef CONFIG_NET_CLS_ROUTE
1447 #ifdef CONFIG_IP_MULTIPLE_TABLES
1448         set_class_tag(rt, fib_rules_tclass(res));
1449 #endif
```

```
1450         set_class_tag(rt, itag);
1451 #endif
1452         rt->rt_type = res->type;
1453 }
```

– La fonction `rt_intern_hash()` est utilisée (ligne 2162) pour placer l'entrée du cache dans la table de hachage. Par ailleurs, cette entrée forme la valeur renvoyée par la fonction.

33.2 Seconde étape : constitution du paquet IP

33.2.1 Implémentation des options IP

Avant d'étudier la fonction `ip_push_pending_frames()`, nous avons besoin de connaître une structure de données supplémentaires, celle correspondant aux options IP. L'envoi d'un paquet IP suppose en principe que toutes les indications nécessaires soient contenues dans l'en-tête IP du paquet. Toutefois, il arrive que des paquets contenant les informations supplémentaires exigées dans l'en-tête de protocole soient émis occasionnellement à fins de diagnostic, ou lorsque l'itinéraire d'un paquet est déjà fixé avant l'émission. Pour ce faire on peut ajouter, comme nous l'avons déjà vu lors de l'analyse de l'en-tête IP, un champ d'option de longueur variable à chaque en-tête IP. L'implémentation Linux utilise une structure de données, `struct ip_options`, définie dans le fichier `linux/include/linux/ip.h`:

Code Linux 2.6.10

```
88  struct ip_options {
89    __u32         faddr;               /* Adresse du premier saut sauvegardee              */
90    unsigned char optlen;
91    unsigned char srr;
92    unsigned char rr;
93    unsigned char ts;
94    unsigned char is_setbyuser:1,      /* Positionne par setsockopt ?                      */
95                  is_data:1,           /* Options dans __data, plutot que dans skb         */
96                  is_strictroute:1,    /* Route source stricte                             */
97                  srr_is_hit:1,        /* L'adresse de destination du paquet etait la notre */
98                  is_changed:1,        /* La somme de controle IP n'est plus valide        */
99                  rr_needaddr:1,       /* Necessite d'enregistrer l'adresse du peripherique
                                            d'entree  */
100                 ts_needtime:1,       /* Necessite d'enregistrer l'estampille temporelle  */
101                 ts_needaddr:1;       /* Necessite d'enregistrer l'adresse du peripherique
                                            de sortie */
102   unsigned char router_alert;
103   unsigned char __pad1;
104   unsigned char __pad2;
105   unsigned char __data[0];
106 };
```

33.2.2 Récupération des fragments

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

Code Linux 2.6.10

```
1107 /*
1108  *      Combine tous les fragments IP en suspens sur la socket en tant que datagramme IP
1109  *      et les pousse dehors.
1110  */
1111 int ip_push_pending_frames(struct sock *sk)
1112 {
1113         struct sk_buff *skb, *tmp_skb;
1114         struct sk_buff **tail_skb;
1115         struct inet_opt *inet = inet_sk(sk);
```

```
1116        struct ip_options *opt = NULL;
1117        struct rtable *rt = inet->cork.rt;
1118        struct iphdr *iph;
1119        int df = 0;
1120        __u8 ttl;
1121        int err = 0;
1122
1123        if ((skb = __skb_dequeue(&sk->sk_write_queue)) == NULL)
1124                goto out;
1125        tail_skb = &(skb_shinfo(skb)->frag_list);
1126
1127        /* deplace skb->data a l'en-tete ip depuis l'en-tete ext */
1128        if (skb->data < skb->nh.raw)
1129                __skb_pull(skb, skb->nh.raw - skb->data);
1130        while ((tmp_skb = __skb_dequeue(&sk->sk_write_queue)) != NULL) {
1131                __skb_pull(tmp_skb, skb->h.raw - skb->nh.raw);
1132                *tail_skb = tmp_skb;
1133                tail_skb = &(tmp_skb->next);
1134                skb->len += tmp_skb->len;
1135                skb->data_len += tmp_skb->len;
1136                skb->truesize += tmp_skb->truesize;
1137                __sock_put(tmp_skb->sk);
1138                tmp_skb->destructor = NULL;
1139                tmp_skb->sk = NULL;
1140        }
1141
1142        /* A moins que l'utilisateur ait demande une decouverte de pmtu reelle
1143         * (IP_PMTUDISC_DO), nous permettons la fragmentation de la trame generee ici.
1144         * Qu'importe que la transformation change la taille du paquet, il sera expulse.
1145         */
1146        if (inet->pmtudisc != IP_PMTUDISC_DO)
1147                skb->local_df = 1;
1148
1149        /* Le bit DF est positionne lorsque nous voulons voir DF sur les trames sortantes.
1150         * Si local_df est egalement positionne, nous permettons encore de fragmenter cette
1151         * trame localement. */
1152        if (inet->pmtudisc == IP_PMTUDISC_DO ||
1153            (!skb_shinfo(skb)->frag_list && ip_dont_fragment(sk, &rt->u.dst)))
1154                df = htons(IP_DF);
1155
1156        if (inet->cork.flags & IPCORK_OPT)
1157                opt = inet->cork.opt;
1158
1159        if (rt->rt_type == RTN_MULTICAST)
1160                ttl = inet->mc_ttl;
1161        else
1162                ttl = ip_select_ttl(inet, &rt->u.dst);
1163
1164        iph = (struct iphdr *)skb->data;
1165        iph->version = 4;
1166        iph->ihl = 5;
1167        if (opt) {
1168                iph->ihl += opt->optlen>>2;
1169                ip_options_build(skb, opt, inet->cork.addr, rt, 0);
1170        }
1171        iph->tos = inet->tos;
1172        iph->tot_len = htons(skb->len);
1173        iph->frag_off = df;
1174        if (!df) {
1175                __ip_select_ident(iph, &rt->u.dst, 0);
1176        } else {
1177                iph->id = htons(inet->id++);
```

```
1178         }
1179         iph->ttl = ttl;
1180         iph->protocol = sk->sk_protocol;
1181         iph->saddr = rt->rt_src;
1182         iph->daddr = rt->rt_dst;
1183         ip_send_check(iph);
1184
1185         skb->priority = sk->sk_priority;
1186         skb->dst = dst_clone(&rt->u.dst);
1187
1188         /* Netfilter recupere tout le skb non fragmente. */
1189         err = NF_HOOK(PF_INET, NF_IP_LOCAL_OUT, skb, NULL,
1190                       skb->dst->dev, dst_output);
1191         if (err) {
1192                 if (err > 0)
1193                         err = inet->recverr ? net_xmit_errno(err) : 0;
1194                 if (err)
1195                         goto error;
1196         }
1197
1198 out:
1199         inet->cork.flags &= ~IPCORK_OPT;
1200         if (inet->cork.opt) {
1201                 kfree(inet->cork.opt);
1202                 inet->cork.opt = NULL;
1203         }
1204         if (inet->cork.rt) {
1205                 ip_rt_put(inet->cork.rt);
1206                 inet->cork.rt = NULL;
1207         }
1208         return err;
1209
1210 error:
1211         IP_INC_STATS(IPSTATS_MIB_OUTDISCARDS);
1212         goto out;
1213 }
```

Autrement dit :

- On déclare un descripteur de tampon de socket, un descripteur de tampon de socket provisoire et une liste de descripteurs de tampon de socket.
- On déclare une option Internet, que l'on initialise avec le champ correspondant du descripteur de couche transport passé en argument.
- On déclare une adresse d'options IP, que l'on initialise à `NULL`.
- On déclare une entrée de cache de routage, que l'on initialise avec celle associée au bouchon de l'option Internet.
- On déclare un en-tête IP, un test pour DF (initialisé à 0), une durée de vie et un code d'erreur (initialisé à 0).
- On essaie de récupérer le premier descripteur de tampon de la file d'attente en écriture du descripteur de couche transport passé en argument. S'il n'y en a pas, c'est qu'il n'y a pas de données à envoyer ; on retire alors le drapeau `IPCORK_OPT` du champ bouchon de l'option Internet, on libère éventuellement les options IP et l'entrée de cache de routage du bouchon et on renvoie 0.

  La constante symbolique `IPCORK_OPT` est définie dans le fichier `linux/include/linux/ip.h` :

Code Linux 2.6.10

```
147 #define IPCORK_OPT      1       /* Des options IP etaient detenues dans ipcork.opt */
```

– On initialise l'adresse de la liste des descripteurs de tampon de socket avec celle du champ liste des fragments du descripteur de tampon que l'on vient de récupérer.
– On change éventuellement l'adresse du début des données du descripteur de tampon pour tenir compte de l'en-tête IP.
– On récupère un à un les descripteurs de tampon de la file d'attente en écriture du descripteur de couche transport passé en argument et on les place dans la liste des fragments (lignes 1130–1140).
– À moins que l'utilisateur n'ait demandé une découverte de pmtu réelle (drapeau IP_PMTUDISC_DO), nous permettons la fragmentation de la trame générée ici.
– Le bit DF est positionné lorsque nous voulons voir DF sur les trames sortantes. Si `local_df` est également positionné, nous permettons encore de fragmenter cette trame localement.

Code Linux 2.6.10

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

```
179 static inline
180 int ip_dont_fragment(struct sock *sk, struct dst_entry *dst)
181 {
182         return (inet_sk(sk)->pmtudisc == IP_PMTUDISC_DO ||
183                 (inet_sk(sk)->pmtudisc == IP_PMTUDISC_WANT &&
184                  !(dst_metric(dst, RTAX_LOCK)&(1<<RTAX_MTU))));
185 }
```

– Si le drapeau `IPCORK_OPT` est positionné parmi les drapeaux du bouchon de l'option Internet, on instantie de descripteur d'options IP avec les options du bouchon de l'option Internet.
– On initialise la durée de vie.

Code Linux 2.6.10

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

```
118 static inline int ip_select_ttl(struct inet_opt *inet, struct dst_entry *dst)
119 {
120         int ttl = inet->uc_ttl;
121
122         if (ttl < 0)
123                 ttl = dst_metric(dst, RTAX_HOPLIMIT);
124         return ttl;
125 }
```

– On instantie l'adresse de l'en-tête IP avec celle du début des données du descripteur de tampon.
– On renseigne les champs de cet en-tête: la version est IPv4, la longueur de l'en-tête est de 5 mots, on place éventuellement les options (ce qui ne nous intéresse pas pour le moment), le type de service est celui indiqué dans l'option Internet, la longueur totale du paquet est celle indiquée dans le descripteur de tampon, le décalage de fragment est positionné, la durée de vie est celle déterminée ci-dessus, le protocole est celui spécifié par le descripteur de couche transport, les adresses source et de destination sont fournies par l'entrée de cache de routage (lignes 1165–1182).

Code Linux 2.6.10

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

```
939 void __ip_select_ident(struct iphdr *iph, struct dst_entry *dst, int more)
940 {
941         struct rtable *rt = (struct rtable *) dst;
942
943         if (rt) {
944                 if (rt->peer == NULL)
```

```
945                         rt_bind_peer(rt, 1);
946
947                 /* Si peer est attache a destination, il n'est jamais detache,
948                    aussi n'avons-nous pas besoin d'attraper un verrou pour le
                      dereferencer.
949                  */
950                 if (rt->peer) {
951                         iph->id = htons(inet_getid(rt->peer, more));
952                         return;
953                 }
954         } else
955                 printk(KERN_DEBUG "rt_bind_peer(0) @%p\n", NET_CALLER(iph));
956
957         ip_select_fb_ident(iph);
958 }
```

– On calcule la somme de contrôle de l'en-tête IP et on la place dans celui-ci.

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

Code Linux 2.6.10

```
95  /* Genere une somme de controle pour un datagramme IP sortant. */
96  __inline__ void ip_send_check(struct iphdr *iph)
97  {
98          iph->check = 0;
99          iph->check = ip_fast_csum((unsigned char *)iph, iph->ihl);
100 }
```

– On renseigne les champs priorité et destination du descripteur de tampon.

– On permet un point d'ancrage `Netfilter` dont la fonction suivante, l'étape qui nous intéresse, est `dst_output()`, que nous allons étudier ci-dessous.

33.3 Troisième étape : transfert de la couche réseau à la couche inférieure

33.3.1 Première sous-étape : détermination de la fonction de transfert

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

Code Linux 2.6.10

```
219 /* Paquet sortant de reseau a transport.  */
220 static inline int dst_output(struct sk_buff *skb)
221 {
222         int err;
223
224         for (;;) {
225                 err = skb->dst->output(skb);
226
227                 if (likely(err == 0))
228                         return err;
229                 if (unlikely(err != NET_XMIT_BYPASS))
230                         return err;
231         }
232 }
```

En général, le pointeur `dst->output` a été initialisé à `ip_output()`, dans le cas d'un paquet, lors de l'étape de redirection, comme nous l'avons vu ci-dessus.

33.3.2 Deuxième sous-étape : fragmentation éventuelle

La fonction `ip_output(skb)` décide si le paquet doit être transmis immédiatement à la fonction `ip_finish()` ou s'il faut au préalable que la fonction `ip_fragment()` l'adapte à la taille d'une trame de la couche liaison.

Code Linux 2.6.10

Cette fonction est définie dans le fichier `linux/net/ipv4/ip_output.c` :

```
287 int ip_output(struct sk_buff *skb)
288 {
289         IP_INC_STATS(IPSTATS_MIB_OUTREQUESTS);
290
291         if (skb->len > dst_pmtu(skb->dst) && !skb_shinfo(skb)->tso_size)
292                 return ip_fragment(skb, ip_finish_output);
293         else
294                 return ip_finish_output(skb);
295 }
```

Nous étudierons la fonction `ip_fragment()` au chapitre 36. Étudions donc la fonction `ip_finish_output()`.

33.3.3 Troisième sous-étape : positionnement du type du paquet

Code Linux 2.6.10

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

```
216 int ip_finish_output(struct sk_buff *skb)
217 {
218         struct net_device *dev = skb->dst->dev;
219
220         skb->dev = dev;
221         skb->protocol = htons(ETH_P_IP);
222
223         return NF_HOOK(PF_INET, NF_IP_POST_ROUTING, skb, NULL, dev,
224                        ip_finish_output2);
225 }
```

Autrement dit :

- On déclare un descripteur de périphérique réseau, que l'on instantie avec celui associé à l'entrée de cache du descripteur de tampon de socket passé en argument.
- Ce descripteur est transmis comme valeur au champ descripteur de périphérique réseau du descripteur de tampon.
- Le type de paquet de la sous-couche 2b, à savoir `ETH_P_IP`, est indiqué dans le champ protocole du descripteur de tampon.
- Le point d'ancrage `NF_IP_POST_ROUTING` est éventuellement traité, renvoyant ensuite à la fonction `ip_finish_output2()`.

33.3.4 Quatrième sous-étape : passage à la couche inférieure

Code Linux 2.6.10

La fonction `ip_finish_output2()` est définie dans le fichier `linux/net/ipv4/ip_output.c` :

```
172 static inline int ip_finish_output2(struct sk_buff *skb)
173 {
174         struct dst_entry *dst = skb->dst;
175         struct hh_cache *hh = dst->hh;
176         struct net_device *dev = dst->dev;
177         int hh_len = LL_RESERVED_SPACE(dev);
178
179         /* Soyons plutot paranoiaque que trop intelligent. */
```

```
180         if (unlikely(skb_headroom(skb) < hh_len && dev->hard_header)) {
181                 struct sk_buff *skb2;
182
183                 skb2 = skb_realloc_headroom(skb, LL_RESERVED_SPACE(dev));
184                 if (skb2 == NULL) {
185                         kfree_skb(skb);
186                         return -ENOMEM;
187                 }
188                 if (skb->sk)
189                         skb_set_owner_w(skb2, skb->sk);
190                 kfree_skb(skb);
191                 skb = skb2;
192         }
193
194 #ifdef CONFIG_NETFILTER_DEBUG
195         nf_debug_ip_finish_output2(skb);
196 #endif /*CONFIG_NETFILTER_DEBUG*/
197
198         if (hh) {
199                 int hh_alen;
200
201                 read_lock_bh(&hh->hh_lock);
202                 hh_alen = HH_DATA_ALIGN(hh->hh_len);
203                 memcpy(skb->data - hh_alen, hh->hh_data, hh_alen);
204                 read_unlock_bh(&hh->hh_lock);
205                 skb_push(skb, hh->hh_len);
206                 return hh->hh_output(skb);
207         } else if (dst->neighbour)
208                 return dst->neighbour->output(skb);
209
210         if (net_ratelimit())
211                 printk(KERN_DEBUG "ip_finish_output2: No header cache and no neighbour!\n");
212         kfree_skb(skb);
213         return -EINVAL;
214 }
```

Autrement dit :

- On déclare une entrée de cache de destination, que l'on instantie avec celle associée au descripteur de tampon de socket passé en argument.
- On déclare une entrée du cache matériel, que l'on instantie avec celle associée à l'entrée de cache de destination.
- On déclare un descripteur de périphérique réseau, que l'on instantie avec celui associé à l'entrée de cache de destination.
- On déclare une longueur d'en-tête matériel, que l'on instantie avec celle que l'on déduit du descripteur de périphérique réseau.
- Si l'espace de tête a une taille inférieure à cette longueur et que la méthode `hard_header()` existe pour le périphérique réseau (rappelons que c'est le cas pour les cartes Ethernet avec la fonction `eth_header()`) :
  - On déclare un nouveau descripteur de tampon de socket, que l'on essaie d'instantier avec le descripteur passé en argument mais avec suffisamment de place dans son espace de tête.
  - Si on ne parvient pas à l'instantier, on libère le descripteur de tampon passé en argument et on renvoie l'opposé du code d'erreur `ENOMEM`.
  - Si un descripteur de couche transport est associé au descripteur de tampon passé en argument, on l'associe également au nouveau descripteur de tampon.

La fonction en ligne `skb_set_owner_w()` est définie dans le fichier `linux/include/netsock.h`:

Code Linux 2.6.10

```
1050 static inline void skb_set_owner_w(struct sk_buff *skb, struct sock *sk)
1051 {
1052         sock_hold(sk);
1053         skb->sk = sk;
1054         skb->destructor = sock_wfree;
1055         atomic_add(skb->truesize, &sk->sk_wmem_alloc);
1056 }
```

- On libère le descripteur de tampon passé en argument que l'on remplace par le nouveau descripteur de tampon.

- Si l'entrée de cache matériel existe:
  - On déclare une longueur d'en-tête matériel alignée.
  - On verrouille l'entrée de cache matériel.
  - On instantie la longueur d'en-tête matériel alignée avec celle déterminée ci-dessus.
  - On copie les données de l'entrée de cache à l'emplacement adéquat du descripteur de tampon.
  - On déverrouille l'entrée de cache matériel.
  - On déplace le début de la zone des données du descripteur de tampon.
  - On fait appel à la fonction d'envoi de trame spécifiée par cette entrée de cache et on renvoie le code fourni par cette fonction.

    Il s'agit de la fonction `dev_queue_xmit()` d'émission d'une trame, étudiée au chapitre 34, comme le montre les instances de `struct neigh_ops` définies dans le fichier `linux/net/ipv4/arp.c`:

Code Linux 2.6.10

```
138 static struct neigh_ops arp_generic_ops = {
139         .family =               AF_INET,
140         .solicit =              arp_solicit,
141         .error_report =         arp_error_report,
142         .output =               neigh_resolve_output,
143         .connected_output =     neigh_connected_output,
144         .hh_output =            dev_queue_xmit,
145         .queue_xmit =           dev_queue_xmit,
146 };
147
148 static struct neigh_ops arp_hh_ops = {
149         .family =               AF_INET,
150         .solicit =              arp_solicit,
151         .error_report =         arp_error_report,
152         .output =               neigh_resolve_output,
153         .connected_output =     neigh_resolve_output,
154         .hh_output =            dev_queue_xmit,
155         .queue_xmit =           dev_queue_xmit,
156 };
```

- Si l'entrée de cache matériel n'est pas définie dans l'entrée de cache de destination mais qu'une station de voisinage y est spécifiée, on fait appel à la fonction d'envoi de trame spécifiée par celle-ci et on renvoie le code fourni par cette fonction.
- Sinon on affiche un message noyau, on libère le descripteur de tampon et on renvoie l'opposé du code d'erreur `EINVAL`.