Programmation réseau avec des prises brutes, en IPv4 et en IPv6

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 16 avril 2009. Dernière mise à jour le 20 avril 2009

http://www.bortzmeyer.org/raw-sockets.html

De nombreuses applications réseau nécessitent de contrôler étroitement le contenu du paquet IP, y compris les parties qui ne sont normalement pas accessibles aux applications. Cela peut se faire en C avec des appels adéquats à `setsockopt()` mais, si on veut vraiment changer la plus grande partie du paquet, il peut être plus simple de fabriquer un paquet IP entièrement « à la main » en utilisant des prises **brutes** (en anglais "*raw sockets*"). Voyons comment on fait en IPv4 et si c'est possible en IPv6.

Un petit avertissement, cher lecteur, ce qui suit est d'assez bas niveau, au sens de « proche de la machine ». L'API des prises offre des mécanismes de programmation bien plus simples que les prises brutes. Même si on veut changer une partie de l'en-tête, par exemple le champ TTL, il existe des techniques plus simples <http://www.bortzmeyer.org/ip-header-set.html> à base de `setsockopt()`. Avec les prises brutes, au contraire, le noyau ne fera plus grand'chose pour vous, vous allez devoir vous débrouiller seul (prononcer le tout avec une voix sépulcrale). Considérez vous mis en garde.

D'abord, le principe, tel qu'illustré depuis de nombreuses années par les applications qui ont besoin d'accéder à ces fonctions de bas niveau, comme traceroute. On déclare la prise de type `SOCK_RAW`, de protocole `IPPROTO_RAW` et, pour bien dire au noyau qu'on se charge de tout, on utilise l'option `IP_HDRINCL` de `setsockopt` :

```
int             on = 1;
...
sd = socket(result->ai_family, SOCK_RAW, IPPROTO_RAW);
setsockopt(sd, IPPROTO_IP, IP_HDRINCL, (char *) &on, sizeof(on));
```

Ensuite, on crée une structure pour représenter le paquet IP (ici, on utilisera UDP) :

```
struct opacket4 {
    struct ip       ip; /* 'struct ip' est une représentation en C
    d'un paquet IP, déclarée dans <netinet/ip.h> */
    struct udphdr   udp;
    char            payload[MSGSIZE];
} __attribute__ ((packed)); /* La directive 'packed' s'assure que le
compilateur ne laissera pas d'espace entre les champs */
...
struct opacket4 op4; /* Output Packet of IP version 4 */
/* S'assurer que tout est à zéro au début */
memset(&op4.ip, '\0', sizeof(op4.ip));
```

Enfin, on peuple cette structure à sa guise, en lisant le RFC 791 [1] pour savoir ce qu'il faut mettre :

```
op4.ip.ip_v = 4; /* Version */
op4.ip.ip_hl = sizeof(op4.ip) >> 2; /* Taille des en-têtes (en IPv4,
                                       elle est variable) */
op4.ip.ip_dst = ... (un champ sin_addr...)
/* Sur Linux, l'adresse source, op4.ip.ip_src, est mise
automatiquement, même pour une prise brute */
op4.ip.ip_p = SOL_UDP; /* Protocole de transport utilisé */
op4.ip.ip_ttl = 1; /* Limite à un seul saut */
headersize = sizeof(op4.ip) + sizeof(op4.udp);
packetsize = headersize + strlen(message);
op4.ip.ip_len = htons(packetsize);
```

On fait pareil pour la partie UDP et on peut alors envoyer le paquet :

```
sendto(sd, &op4, packetsize, 0, result->ai_addr,
               result->ai_addrlen);
```

Le programme complet (en ligne sur `http://www.bortzmeyer.org/files/send-udp-with-raw-socket.c`) qui met en œuvre cette technique s'utilise ainsi :

```
# ./send-udp -v --port 5354 192.168.2.1
Connecting to 192.168.2.1...
Sending 42 bytes (28 for the headers)...
```

et, vu avec tcpdump, cela donne bien le résultat attendu :

```
13:55:12.241391 IP (tos 0x0, ttl 1, id 44977, offset 0, flags [none], \
     proto UDP (17), length 42) 192.168.2.2.5354 > 192.168.2.1.5354: \
     [no cksum] UDP, length 14
```

L'adresse source a été mise seule mais le programme a une option pour la forcer :

1. Pour voir le RFC de numéro NNN, `https://www.ietf.org/rfc/rfcNNN.txt`, par exemple `https://www.ietf.org/rfc/rfc791.txt`

```
# ./send-udp -v --port 5354 --source 192.0.2.1 192.168.2.1
Connecting to 192.168.2.1...
Sending 42 bytes (28 for the headers)...
```

qui donne :

```
13:55:24.815620 IP (tos 0x0, ttl 23, id 45015, offset 0, flags [none], \
     proto UDP (17), length 42) 192.0.2.1.5354 > 192.168.2.1.5354: \
     [no cksum] UDP, length 14
```

Il existe quelques pièges, par exemple le fait que la somme de contrôle de l'en-tête IP (cf. RFC 791) n'est pas forcément remplie automatiquement (ça dépend du noyau utilisé) et qu'il peut donc être nécessaire de la calculer soi-même. Je l'ai dit, avec les prises brutes, il faut penser à tout.

OK, ça, c'était pour IPv4. Et en IPv6 ? Eh bien, c'est plus compliqué. Pour des raisons exposées dans la section 1 du RFC 3542, l'option `IP_HDRINCL` n'est pas garantie en IPv6. La démarche « normale » est d'utiliser les techniques dudit RFC, à base de `setsockopt` et de renoncer aux prises brutes (attention, le RFC utilise, en sa section 3, le terme de "*raw sockets*" quand même, pour désigner les prises où un appel à `setsockopt` a permis de modifier certains champs). (La section 5 du RFC 3542 propose une autre approche dite d'"*ancillary data*".)

Si on veut quand même faire de la prise brute en IPv6, il n'y a que des solutions non portables. On peut se servir du BPF (où on peut modifier tout ce qui est au dessus de la couche 2, option `PF_PACKET` de `socket()`, cf. `packet`(7)) ou bien utiliser les prises brutes comme en IPv4, sur les systèmes où ça marche, typiquement Linux.

(Mathieu Peresse me fait remarquer que, sur Linux, « après examen du code du noyau, pour IPv4, créer une RAW socket IPv4 avec IPPROTO_RAW comme protocole équivaut exactement à créer une socket RAW IPv4 et mettre l'option IP_HDRINCL à 1 (la fonctionnalité d'inclure le header est en fait activée en mettant le membre "hdrincl" de la structure "inet_sock" à 1)... ».)

Sur FreeBSD et NetBSD, `IP_HDRINCL` ne marche tout simplement pas (cf. <`http://lists.freebsd.org/pipermail/freebsd-pf/2006-May/002174.html`>).

Le programme indiqué plus haut (en ligne sur `http://www.bortzmeyer.org/files/send-udp-with-raw-soc` `c`) marche donc également en IPv6 mais sur Linux uniquement. La prise est créée avec exactement les mêmes options qu'en v4, la configuration en vraie prise brute est légèrement différente, on n'utilise pas `IP_HDRINCL`, juste le type `IPPROTO_RAW`. Et les structures de données sont proches :

```
struct opacket6 {
    struct ip6_hdr  ip;
    struct udphdr   udp;
    char            payload[MSGSIZE];
} __attribute__ ((packed));
struct opacket6 op6;
```

sauf que cette fois, il faut lire le RFC 2460 en les remplissant (des champs comme "*Protocol*" ou "*TTL*" n'ont pas le même nom, car leur sémantique est légèrement différente) :

```
memset(&op6.ip, '\0', sizeof(op6.ip));
op6.ip.ip6_vfc = 6 << 4; /* Les quatre premiers bits stockent la
                                  version, ici 6 */
op6.ip.ip6_dst = sockaddr6->sin6_addr;
op6.ip.ip6_nxt = SOL_UDP;
op6.ip.ip6_hlim = TTL;
headersize = sizeof(op6.ip) + sizeof(op6.udp);
packetsize = headersize + strlen(message);
op6.ip.ip6_plen = htons((uint16_t) packetsize);
```

Par défaut, si la machine visée a des adresses v4 et v6, il utilise la politique de préférence du système (celle qui, sur Linux, se règle dans `/etc/gai.conf`) , mais on peut la forcer avec les options `-4` et `-6`. tcpdump voit :

```
14:07:52.030339 ::.5354 > 2a01:e35:8bd9:8bb0:a00:20ff:fe99:faf4.5354: \
               UDP, length: 14 (len 62, hlim 1)
```

On note que l'adresse source n'a pas été mise automatiquement, en IPv6. Elle reste donc à zéro (`::`, qui veut dire « adresse indéfinie »). Si on la force :

```
14:08:29.745203 2001:db8::1.5354 > 2a01:e35:8bd9:8bb0:a00:20ff:fe99:faf4.5354: \
              UDP, length: 14 (len 62, hlim 1)
```

Merci à Mathieu Peresse et à Yves Legrand-Gérard pour leur aide. Outre le RFC, on peut aussi trouver une bonne explication en français de l'API dans le livre de Gisèle Cizault <`http://livre.point6.net/index.php/Programmation_avance`>.