

Tester quels bits de l'en-tête IP on peut changer

Stéphane Bortzmeyer

<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 12 Février 2008

<http://www.bortzmeyer.org/ip-header-set.html>

Le protocole IP permet d'indiquer dans l'en-tête des paquets plusieurs options, qui peuvent indiquer au routeur un traitement spécifique du paquet. Pour un programme ordinaire, utilisant les API standard, quelles options peuvent être facilement définies à l'émission, et lues à la réception ?

La question a surgi dans le groupe de travail Behave de l'IETF (<http://www.bortzmeyer.org/behave-wg.html>), à propos du protocole TURN (depuis normalisé dans le RFC 5766¹). TURN relaie des paquets, notamment UDP entre deux clients qui ne peuvent pas se parler directement. Le serveur TURN doit (<http://www.bortzmeyer.org/2119.html>)-il respecter les en-têtes des paquets IP ? Si le RFC impose cette contrainte, pourra-t-on programmer un serveur TURN en n'utilisant que les API standard ? Si le serveur tourne sur Unix, devra-t-il le faire en étant root ?

Les bits en question sont définis dans divers RFC. Ainsi, le RFC 791, qui définissait IPv4 à l'origine, définit un groupe de bits nommé le **TOS** ("*Type Of Service*"), qui permet de fixer la priorité du paquet et ses choix (privilégier le débit, le coût, etc). Ce groupe a été remplacé par le **DSCP** ("*Differentiated Services Code Point*") dans le RFC 2474. Puis le RFC 3168 a utilisé deux autres bits pour **ECN** ("*Explicit Congestion Notification*"). D'autres bits sont modifiables potentiellement comme le TTL ou comme **DF** ("*Don't Fragment*").

Mais une application typique ne fabrique pas les paquets IP à la main, bit par bit. Elle compte sur une API standard, typiquement sur Posix. (Sur un système comme Unix, faire des paquets entièrement soi-même et les envoyer nécessite d'être root de toute façon.) Que nous permet cette API ?

C'est pour répondre à cette question que j'ai développé un ensemble de petits programmes, qu'on peut récupérer dans le fichier (en ligne sur <http://www.bortzmeyer.org/files/tests-socket.tar.gz>). Ces programmes sont prévus pour être compilés et exécutés sur un grand nombre de machines et ils affichent un rapport détaillé qui peut être transmis au groupe de travail.

Pour les compiler et les exécuter, voir les instructions dans le fichier README qui est joint. Voici des exemples d'exécution du programme `ip-header-set` qui teste s'il peut demander avec succès les options :

1. Pour voir le RFC de numéro NNN, <http://www.ietf.org/rfc/rfcNNN.txt>, par exemple <http://www.ietf.org/rfc/rfc5766.txt>

```
% ./ip-header-set
Testing of the abilities to set bits in the IP header
My UID is 1000
The system is Linux 2.6.18-5-686, the machine is i686
TTL successfully set to 10
TOS successfully set to 8
ECN successfully set to 2
[PROBLEM] Cannot set the precedence to 160: (Operation not permitted)
DF set through PMTU successfully set to 2
DF clear through PMTU successfully set to 0

Warning: we only tested that setsockopt was happy,
        not that the actual packet headers were changed
```

On voit que, sur Linux, on peut apparemment presque tout fixer à la valeur souhaitée, sauf la priorité qu'on ne peut pas augmenter (si on est root, ça marche). Sur un autre système, les succès et les échecs ne sont pas les mêmes :

```
Testing of the abilities to set bits in the IP header
My UID is 54131
The system is FreeBSD 6.2-RELEASE, the machine is i386
TTL successfully set to 10
TOS successfully set to 8
ECN successfully set to 2
precedence successfully set to 160
No known way to set the DF bit on this system

Warning: we only tested that setsockopt was happy,
        not that the actual packet headers were changed
```

Donc, la fonction de définition des options a marché. Mais le paquet IP émis a t-il réellement changé ? Pour le savoir, nous utilisons désormais deux programmes, `sender` qui va fixer les options **et** envoyer les paquets **et** `receiver` qui va les lire. `sender` affiche à peu près la même chose que `ip-header-set` mais il envoie réellement le paquet UDP :

```
% ./sender test.example.org 10000
Testing of the abilities to set bits in the IP header
My UID is 1000
The system is Linux 2.6.18-5-686, the machine is i686
TTL successfully set to 10
TOS successfully set to 8
ECN successfully set to 2
[PROBLEM] Cannot set the precedence to 160:
DF set through PMTU successfully set to 2
DF clear through PMTU successfully set to 0
```

et, sur la machine de réception (ici, identique) :

```
% ./receiver 10000
Testing of the abilities to read bits in the IP header
My UID is 1000
The system is Linux 2.6.18-5-686, the machine is i686
Received 5 bytes
    The TTL is 10
    The TOS/DSCP is 0
Received 5 bytes
    The TTL is 64
    The TOS/DSCP is 8
```

```

Received 5 bytes
  The TTL is 64
  The TOS/DSCP is 2
Received 5 bytes
  The TTL is 64
  The TOS/DSCP is 0
Received 5 bytes
  The TTL is 64
  The TOS/DSCP is 0
Received 5 bytes
  The TTL is 64
  The TOS/DSCP is 0

```

On voit que l'API disponible sur Linux permet de lire les options, et qu'elles ont la bonne valeur. Si on n'a pas confiance, on peut vérifier avec un "sniffer" sniffer comme tcpdump :

```

09:44:00.127838 IP (tos 0x0, ttl 10, id 35166, offset 0, flags [DF], proto: UDP (17), length: 33) 192.0.2.69.47
09:44:00.127873 IP (tos 0x8, ttl 64, id 35166, offset 0, flags [DF], proto: UDP (17), length: 33) 192.0.2.69.47

```

Sur cette trace des deux premiers paquets, on voit le changement du TTL à 10, et on voit le TOS mis à 8.

Comment travaillent ces programmes ? Pour définir les options comme le TOS/DSCP ou comme ECN, ils utilisent l'appel système `setsockopt` ainsi :

```

myecn = 0x02; /* ECN-capable transport */
result = setsockopt(mysocket, IPPROTO_IP,
                   IP_TOS, myecn, sizeof(int));

```

On note que Posix appelle toujours TOS (cf. la constante `IP_TOS`) ce qui se nomme normalement DSCP depuis le RFC 2474 il y a neuf ans. Ce n'est pas facile de faire évoluer une API ! Cela explique un certain manque de rigueur dans mes programmes, où TOS et DSCP sont utilisés simultanément.

Pour fixer le TTL, le principe est le même. Mais pour mettre le bit DF ("*Don't Fragment*", voir les RFC 1191 et RFC 2923) à un, il faut le faire indirectement, en demandant d'activer ou non la découverte de la MTU du chemin (et cela ne marche que sur Linux) :

```

result = setsockopt(mysocket, IPPROTO_IP,
                   IP_MTU_DISCOVER, IP_PMTUDISC_DO, sizeof(int));

```

Et pour lire les options ? C'est nettement plus compliqué. Une méthode assez peu standard existe, qui dépend des messages de contrôle des prises (cf. `cmsg`) et de `recvmsg` :

```

const int      One = 1;
...
/* On demande à *recevoir* les messages de contrôle avec le TTL */
result = setsockopt(mysocket, IPPROTO_IP, IP_RECVTTL, &One, sizeof(One));
...
/* Et, pour chaque paquet reçu */
memset(message, 0, sizeof(*message));

```

```
message->msg_iov = malloc(sizeof(struct iovec));
/* Les données seront mises dans "buffer" */
message->msg_iov->iov_base = buffer;
message->msg_iov->iov_len = length;
message->msg_iovlen = 1;
/* Et les messages de contrôle dans "control_buffer" */
message->msg_control = control_buffer;
message->msg_controllen = control_length;
result = recvmsg(mysocket, message, 0);
if (result <= 0) {
err_ret("[PROBLEME] Cannot receive data (%s)", sys_err_str());
} else {
fprintf(stdout, "Received %i bytes\n", result);
for (c_msg = CMSG_FIRSTHDR(message); c_msg;
      c_msg = (CMSG_NXTHDR(message, c_msg))) {
if (c_msg->cmsg_level == IPPROTO_IP &&
      c_msg->cmsg_type == IP_TTL) {
fprintf(stdout, "\tThe TTL is %i\n",
*(int *) CMSG_DATA(c_msg));
}
```

Merci au regretté W. Richard Stevens pour son excellent livre sur la programmation réseau sur Unix (<http://www.bortzmeyer.org/unix-network-programming.html>).