Programmation réseau 5

Juliusz Chroboczek

24 février 2019

TCP est une couche épaisse au dessus de IP, et implémente des fonctionnalités qui ne sont pas forcément désirables pour toute application :

- TCP fournit un service *fiable*, et, en cas de perte, reémet des données qui peuvent être obsolètes ;
- TCP fournit un service *ordonné*, et retarde donc les paquets qui sont arrivés en avance (qui ont doublé un autre paquet, ou qui suivent un paquet qui doit être réémis) ;
- TCP implémente un service de communication par *flots*, et peut donc diviser une écriture en plusieurs paquets, ou, a contrario, agréger plusieurs écritures en un seul paquet ;
- TCP est contrôlé, ce qui réduit la flexibilité qu'a l'application à gérer le débit elle-même (par exemple en changeant la résolution ou le codage d'un flot vidéo en cas de congestion).

Ces fonctionnalités ne peuvent pas être éteintes individuellement : une application qui utilise TCP profite forcément de toutes. Pour les éviter, il faut utiliser un autre protocole de couche transport.

1 Le protocole UDP

UDP est tout le contraire de TCP : il s'agit de la couche la plus fine possible au dessus de IP. Essentiellement, UDP n'ajoute que les numéros de ports à IP, ce qui permet à une application de l'utiliser (un paquet IP est destiné à une interface, pas à une *socket* au sein d'une machine).

L'unité de la communication en UDP s'appelle un *datagramme*. UDP n'offre que deux fonctionnalités : l'envoi et la réception d'un datagramme. L'envoi d'un datagramme entraîne l'émission d'un paquet IP (sans aucune retransmission, segmentation ou réémission), et la réception d'un paquet IP contenant un datagramme entraîne la réception de ce dernier.

2 Programmation UDP

La communication UDP se fait vers ou depuis une *socket* UDP. Cette *socket* est créée à l'aide de l'appel système `socket` avec le paramètre `type` valant `SOCK_DGRAM`. L'appel `bind` s'applique à une *socket* UDP comme en TCP, et sert à définir le numéro de port source des datagrammes sortants, ainsi qu'à sélectionner les datagrammes entrants par leur numéro de port destination. Enfin, on utilise les appels système `sendto` et `recvfrom` pour émettre et recevoir des datagrammes.

UDP est beacoup plus flexible que TCP : il peut être utilisé en suivant une structure client-serveur, mais permet aussi de faire du pair-à-pair pur. Dans ce cours, nous utiliserons UDP pour faire du client-serveur ; le projet utilisera une structure pair-à-pair.

Dans ce qui suit, je décris la structure d'un client et d'un serveur UDP simplistes ; comme nous le verrons par la suite, il manque à ces programmes plusieurs fonctionnalités essentielles.

2.1 Client UDP simpliste

Du côté client, on commence par créer une *socket* :

```
s = socket(AF_INET6, SOCK_DGRAM, 0);
if(s < 0) ...
```

Comme on est du côté client, le port source est indifférent, pas la peine d'appeler `bind`. On construit une `struct sockaddr` correspondant à l'adresse de *socket* serveur, et on envoie notre requête :

```
unsigned char req[reqlen];
/* construire la requête */
struct sockaddr_sin6 server;
/* remplir la sockaddr */
rc = sendto(s, req, reqlen, 0,
            (struct sockaddr*)&server, sizeof(server));
if(rc < 0)
    ...
```

L'envoi d'un datagramme est une opération atomique, c'est-à-dire qui réussit ou qui échoue dans sa totalité ; il n'est donc pas nécessaire de gérer les écritures partielles. On attend maintenant une réponse :

```
unsigned char reply[4096];
rc = recvfrom(s, reply, 4096, 0, NULL, NULL);
if(rc < 0)
    ...
```

On a reçu notre réponse, on peut faire le ménage :

```
close(s);
```

2.2 Serveur UDP simpliste

On commence par créer la *socket* :

```
s = socket(AF_INET6, SOCK_DGRAM, 0);
```

Comme on est le serveur, il faut la lier à un port :

```
struct sockaddr_in6 server;
memset(&server, 0, sizeof(server))
server.sin6_family = AF_INET6;
server.sin6_port = htons(4242);
rc = bind(s, (struct sockaddr*)&server, sizeof(server));
if(rc < 0)
    ...
```

La boucle principale du serveur lit une requête et envoie une réponse :

```
while(1) {
    unsigned char req[4096], reply[4096];
    struct sockaddr_in6 client;
    int client_len = sizeof(client);
    rc = recvfrom(s, req, 4096, 0, &client, &client_len);
    if(rc < 0) {
        perror("recvfrom");
        continue;
    }
    /* analyser la requête, construire la réponse */
    rc = sendto(s, reply, reply_len, 0, &client, &client_len);
}
```

Remarquez que l'adresse du client est simplement obtenue lors de l'appel `recvfrom`, qui nous indique l'adresse source du datagramme reçu. Remarquez aussi que le dernier paramètre de `recvfrom` est un paramètre *in-out* : en entrée, il indique la taille maximale de l'adresse, en sortie, il indique sa taille effective.

3 Limitations

Les deux programmes décrits ci-dessus sont très limités. En production, il faudra prendre soin de gérer un certain nombres de détails que TCP gère automatiquement.

Du côté client, il faut s'assurer qu'on n'accepte pas de réponses à des requêtes qu'on n'a pas envoyées. Le strict minimum est d'analyer l'adresse retournée par `recvfrom` pour vérifier que c'est bien celle du serveur. Si on contrôle la couche application, il est bon d'inclure dans le datagramme lui-même un *nonce*[1], une valeur tirée au hasard (et donc difficile à deviner) qui sera retournée par le serveur et vérifiée par le client[2].

Du côté client toujours, il faut gérer la possibilité que la requête soit perdue ou que la réponse le soit. Si le protocole est conçu de façon que les requêtes soient idempotentes (on peut les exécuter plusieurs fois sans mauvais effets), alors il suffit d'attendre la réponse un temps fini puis de la réémettre (par exemple avec un *backoff exponentiel*) ; nous verrons au cours suivant comment faire celà avec l'appel système `select`.

1. Je propose *hapax*.
2. Dans le protocole NTP, le temps d'émission sert de *nonce*.

Du côté serveur, nous avons un problème plus embêtant. Si le serveur a plusieurs adresses (par exemple parce qu'il est *multi-homed*), il se peut que l'appel système `sendto` sélectionne une adresse différente de celle ou la requête a été reçue ; le datagramme contenant la réponse sera alors rejeté par le client. Nous verrons par la suite comment utiliser les messages de contrôle pour utiliser la bonne adresse source.

Enfin, comme UDP ne gère pas la segmentation, il faut s'assurer que le datagramme émis ne dépasse pas le PMTU, c'est à dire la plus petite taille maximale de paquet (MTU) de la route utilisée. IPv6 garantit que le PMTU fait au moins 1280 octets ; on peut donc éviter de calculer le PMTU si on se limite à une charge de 1024 octets. Nous verrons par la suite comment utiliser la fragmentation de couche réseau (ce qui est mal) ou calculer le PMTU (ce qui est bien) pour dépasser cette limite.