Programmation réseau 6

Juliusz Chroboczek

25 février 2019

Le client naïf vu au cours précédent a deux problèmes : il bloque en cas de perte de paquet, et il accepte des réponses à des requêtes qu'il n'a jamais envoyées.

1 Gestion des *timeouts*

UDP est non-fiable, il se peut donc que la requête ou que la réponse soient perdues. Si une requête est perdue, il suffit de la reémettre après un *timeout*.

Le cas de la réponse qui est perdue est, du côté client, impossible à distinguer du cas où c'est la requête qui est perdue. Si la requête était idempotente (l'exécuter plusieurs fois a le même effet que l'exécuter une seule fois), alors il suffit de la reémettre, comme dans le cas d'une requête perdue. Si par contre elle ne l'est pas, il faut une collaboration du serveur ; il est donc souhaitable d'utiliser des protocoles de couche application où les requêtes sont idempotentes[1].

Deux techniques sont utilisées pour implémenter un *timeout*. Tout d'abord, on utilisera des descripteurs de fichier non-bloquants, sur lesquels aucun appel système ne bloque. Ensuite, on utilisera l'appel système `select` pour attendre une donnée. Ces techniques sont décrites en détail dans la partie 4 de mon poly de programmation système.

On commence par créer une *socket*, puis la mettre en mode non-bloquant :

```
s = socket(AF_INET6, SOCK_DGRAM, 0);
if(s < 0) ...
rc = fcntl(s, F_GETFL);
if(rc < 0) ...
rc = fcntl(s, F_SETFL, rc | O_NONBLOCK);
if(rc < 0) ...
```

On se fixe un *timeout*, puis on envoie la requête :

```
int to = 1;
struct sockaddr_in6 server;
unsigned char req[req_len];
/* remplir la sockaddr et la requête */
```

1. Par exemple, au lieu de dire « crée un fichier et retourne son nom », on dira « crée un fichier nommé `toto` sauf s'il existe déjà ».

```
again:
  rc = sendto(s, req, req_len,
              (struct sockaddr*)&server, sizeof(server));
  if(rc < 0) {
      if(errno == EAGAIN) {
          sleep(to);
          goto again;
      perror("sendto");
      exit(1);
  }
```

Si les tampons d'émission de la *socket* étaient pleins, l'appel système `sendto` échouera avec `errno` valant `EAGAIN` au lieu de bloquer ; on pourrait faire les choses proprement (à coup de `select`), mais les marges de ce poly sont trop étroites, donc on attend un moment puis on reessaie. Sinon, il y a eu une « vraie » erreur, ce serait probablement peine perdue de reessayer.

Attendons la réponse :

```
fd_set readfds;
FD_ZERO(&readfds);
FD_SET(s, &readfds);
struct timeval tv = {to, 0};
rc = select(s + 1, &readfds, NULL, NULL, &tv);
if(rc < 0) {
    perror("select");
    exit(1);
}
```

Si une réponse est arrivée, alors `rc` vaut 1, et le descripteur `s` est dans `readfds`. Sinon, on augmente le *timeout* et on recommence :

```
if(rc > 0) {
    if(FD_ISSET(s, &readfds)) {
        unsigned char reply[4096];
        rc = recvfrom(s, reply, 4096, 0, NULL, NULL);
        if(rc < 0) {
            if(errno == EAGAIN) {
                goto again;
            } else {
                perror("recvfrom");
                exit(1);
            }
        }
        /* réponse bien reçue, on peut la gérer */
    } else {
```

```
            fprintf(stderr,
                    "Eek !  Descripteur de fichier inattendu !\n");
            exit(1);
        }
    } else {
        /* timeout */
        if(to < 64) {
            to = to * 2;        /* backoff exponentiel */
            goto again;
        } else {
            fprintf(stderr, "Timeout dépassé.\n");
            exit(1);
        }
    }
```

Il y a plusieurs points à commenter. Tout d'abord, on vérifie quel descripteur de fichier est présent dans `readfds` ; dans ce programme, il n'y en a qu'un, mais un programme plus général pourrait communiquer avec plusieurs serveurs simultanément, il faudrait alors parcourir tous les descripteurs possibles et tester chacun à l'aide de `FD_ISSET`.

Ensuite, on commence par un *timeout* très faible (1 s dans cet exemple) qu'on multiplie par deux à chaque fois, ce qui nous permet de gérer rapidement un premier paquet perdu sans pour autant trop surcharger un serveur lent ; cet algorithme s'appelle le *backoff exponentiel*. (Dans un programme qui envoie plusieurs requêtes au même serveur, on pourrait commencer par faire un *backoff exponentiel*, et, à partir de la deuxième requête, initialiser le *timeout* à une valeur dérivée du temps de réaction du serveur). Au bout de 6 requêtes (`to` égal à 64 s), on abandonne — si un serveur n'a pas répondu au bout de 127 s, c'est qu'il est mort ou alors très mal en point.

Enfin, on met la *socket* en mode non-bloquant, et on gère le cas où `recvmsg` retourne `EAGAIN` et cela alors même que `select` a indiqué que le descripteur était prêt. En effet, un datagramme UDP peut être perdu à tout moment, même s'il est déjà dans les tampons de réception — notamment entre `select` et `recvmsg`. Sans cette procédure, le client souffrirait d'un *deadlock* dans ce cas.

2 Appariement requête-réponse

Le client ci-dessus accepte toute réponse, même si elle ne correspond à aucune requête, ce qui peut causer des confusions si un datagramme est reçu sur le même port, soit par erreur soit du fait d'un attaque. Pour s'en prémunir, il existe deux techniques complémentaires : la vérification de l'adresse de l'émetteur de la réponse, et l'utilisation d'un *nonce* inclus dans les données de couche application. Ces techniques ne sont pas entièrement sécurisées en présence d'un attaquant capable d'intercepter des requêtes — en présence d'un tel attaquant, il faut utiliser des techniques cryptographiques.

2.1 Vérification de l'adresse source

L'appel `recvfrom` nous donne l'adresse source du paquet reçu, dont nous nous sommes servis du côté serveur. Du côté client, on peut s'en servir pour vérifier que le datagramme est en apparence émis par le bon serveur. Remarquez cependant que le protocole IP ne sécurise pas l'adresse d'émission, et qu'il est généralement possible pour un attaquant décidé d'émettre un paquet avec une adresse d'émission usurpée, et cela pas seulement s'il est sur le lien local.

Pour vérifier l'adresse source, il suffit de la comparer bit-à-bit à celle du serveur :

```
unsigned char reply[4096];
struct sockaddr_in6 sin6;
int sin6_len = sizeof(sin6);
rc = recvfrom(s, reply, 4096, 0, &sin6, &sin6_len);
if(rc < 0) ...
if(sin6_len != sizeof(sin6) ||
   sin6.sin6_family != AF_INET6 ||
   memcmp(sin6.sin6_addr, server.sin6_addr, 16) != 0 ||
   sin6.sin6_port != server.sin6_port) {
    fprintf(stderr, "Paquet inattendu.\n");
    goto again;
}
```

2.2 Utilisation d'un *nonce*

Comme il est possible d'usurper une adresse d'émission, la technique ci-dessus ne suffit pas. Le protocole de couche d'application peut prévoir l'utilisation d'un *nonce* : une valeur arbitraire qui est incluse dans la requête et renvoyée dans la réponse. Toute valeur qui est difficile à deviner pour un attaquant convient ; ma préférence personnelle est d'uiliser un *nonce* de 64 bits tiré au hasard à l'aide d'un générateur de nombres aléatoires fort[2].

Si le protocole de couche d'applicaion ne prévoit pas l'inclusion d'un *nonce*, une protection plus faible peut être obtenue en tirant au hasard le numéro de port utilisé du côté client ; les systèmes modernes font ça automatiquement si `bind` n'est pas utilisé (les systèmes plus anciens utilisent des numéros de port séquentiels). Cependant, il est souvent le cas qu'on puisse utiliser un champ existant du protocole comme *nonce* ; par exemple, dans NTP on peut coder un *nonce* aléatoire dans les bits d'ordre bas du *timestamp* de transmission[3].

2.3 Utilisation d'un *challenge* cryptographique

Un *nonce* suffisamment grand est parfaitement sécurisé si l'attaquant n'a pas accès à la requête ; cependant, il se peut que l'attaquant puisse lire la requête. Dans ce cas, il faut utiliser des techniques cryptographiques pour authentifier le serveur auprès du client.

2. `man 7 random`

3. Une technique plus douteuse consiste à coder un nonce dans les bits de valeur 0x20 des noms transmis dans une requête DNS.

Dans la technique du *challenge* cryptographique, le client et le serveur ont une clé partagée. Le client envoie sa requête comme d'habitude, en incluant un *nonce* (ce qui empêche l'attaquant de rejouer une vieille requête correctement signée). Le serveur inclut dans sa réponse non seulement un *nonce*, mais aussi une signature de la réponse complète (*nonce* inclus), obtenue par exemple à l'aide d'un *HMAC* calculé à l'aide d'une fonction de hachage cryptographique.

Remarquez qu'il existe des protocoles cryptographiques standard qui garantissent non seulement l'authenticité et l'intégrité des données transmises, mais aussi leur confidentialité. Cependant, ils sont typiquement adaptés à TCP (TLS, ssh), et ceux qui sont utilisables avec UDP utilisent un *handshake* cryptographique préalable à la commmunication (DTLS). Pour un protocole requête-réponse léger, je ne connais pas d'alternative à l'utilisation d'un mécanisme cryptographique *ad hoc*.