TP 6 : Transfert simple de fichier

Université Paris 7 — LMI&EIDD — Systèmes et Réseaux

Ce TP n'est pas noté

Le but de ce TP est de faire un transfert de fichier par une socket TCP, semblable à la commande `scp` d'Unix.

1 Déroulement d'une session

1. Le serveur est lancé par une commande telle que `./scpd 2000`
2. Le serveur se place en attente de connection sur un port donné (dans cet exemple le port 2000) de sa machine
3. Le client est lancé (depuis la même machine ou une autre) par une commande de type `./scp localhost 2000 truc.txt`
4. Le client se connecte au serveur par une socket TCP (ici sur l'adresse IP `127.0.0.1:2000`)
5. Le client envoie un nom de fichier (nom local à la machine du serveur, bien sûr)
6. Le serveur ouvre le fichier en lecture
7. Le serveur envoie le fichier sur la socket
8. Parallèlement, le client le lit sur la socket et le place dans un fichier local (ayant le même nom que le fichier distant)

Le serveur doit être persistant : à la fin d'un transfert de fichier, il ferme la socket de trasnfert et le fichier, et se replace en attente d'une connection client. Le client lui n'ouvre qu'une socket et se termine à la fin du transfert.

2 Du côté du client

Les actions faites par le client `scp` sont :

1. Verifier que les arguments sont corrects
2. Ouvrir le socket (comme dans le TP5) grâce aux deux premiers arguments de la ligne de commande. Tester les erreurs (signalant l'inexistence du serveur)
3. Écrire dans cette socket le nom de fichier (le 3ème argument)
4. Ouvrir le fichier destination (le 3ème argument aussi) en écriture
5. Par une boucle (ressemblant au `cat` du TP1) lire dans la socket et écrire dans le fichier
6. Fermer le fichier et la socket de communication

Attention il faut tester la valeur de retour de `read()`. Contrairement au TP 1 en effet, si l'on demande à lire x octats et qu'on en reçoit y, avec $0 < y < x$, c'est que la socket TCP a fractionné les données, pas que l'on est au bout du fichier ! Par contre `read()` renvoit 0 si et seulement si la socket a été fermée de l'autre côté, car s'il n'y a rien à lire `read()` bloque.

3 Du côté du serveur

Il faut comme pour le TP4 déclarer une socket en utilisant `socket( )` puis `bind( )` puis `listen( )`. Ensuite le serveur répète inlassablement :

1. Attendre une connection par `accept( )`
2. Sur la nouvelle socket, lire une ligne (de, disons, 1000 caractères maximum) : le nom de fichier
3. Ouvrir ce fichier en lecture. Si échec d'ouverture la socket est fermée immédiatement.
4. L'envoyer sur la socket de communication (par une boucle de `read( )` dans le fichier et `write( )` dans la socket)
5. Fermer le fichier et la socket de communication
6. Goto 1

4 Extensions

Pour améliorer ce protocole on peut implémenter :

4.1 Politique de sécurité du serveur

Vous remarquerez qu'aucune authentification (login/password) n'est demandée. Or le client peut demander n'importe quel fichier auquel vous-même avez accès ! Pour éviter cela il faut au minimum vérifier que le fichier demandé existe, est régulier (pas un répertoire, etc) et que **tous** les usagers on droit de lecture (un `ls -l` doit faire apparaître un `r` en 8ème position).

Cela peut se faire par un appel à `stat( )` (du côté serveur). Il faut alors
- Tout d'abord `stat( )` doit retourner 0 (sinon il y a une erreur, vraissemblablement que le fichier n'existe pas).
- Le fichier doit être régulier : `S.st_mode & S_IFREG` est non nul, si `S` est la `struct stat` remplie par `stat( )`
- Le fichier doit être en lecture pour tous : `S.st_mode & S_IROTH` est non nul

Vous pouvez en plus avoir une politique de sécurité plus stricte, du genre : le fichier doit se trouver dans le répertoire `/pub` (comme FTP), etc.

4.2 Messages d'erreur

Actuellement, en cas d'erreur (fichier inexistant, mauvais droits...) la socket est fermée par le serveur. Le client croit alors qu'il reçoit un fichier vide, et va le créer comme tel. Une solution pour lutter contre cela est d'utiliser un *code de statut*. C'est un octet qui est envoyé en premier, avant le fichier (il faut donc le lire spécifiquement, ce qui demande de modifier le client *et* le serveur !). Typiquement 0 si tout va bien (le fichier est ensuite envoyé) et une valeur non nulle en cas d'erreur, par exemple 1 si fichier inexistant, 2 si mauvais droits d'acces, etc

4.3 Vérification des erreurs

Réfléchir à un mécanisme permettant de détecter d'éventuelles erreurs de transmission. Pour cela le serveur envoie (d'une façon à prévoir dans le protocole !) un code (MD5, CRC32...) que le client revérifie. Il n'y a malheureusement pas de fonction de la bibliotheque standard qui fasse cela, mais des utilitaires en ligne de commande : `crc32` ou `md5sum`.