Les précédents TP vous ont permis de comprendre les bases sur lesquelles s'appuie toute communication utilisant TCP ou UDP. L'objectif du présent TP est de mettre en application les connaissances acquises pour réaliser des transferts de fichiers en utilisant TCP et traiter en parallèle plusieurs clients (serveur concurrent multi-processus).

1 Transfert de fichier avec TCP

L'objectif de cet exercice est de programmer un transfert FTP en utilisant TCP et comprendre les bases sur lesquelles s'appuie l'implémentation du protocole applicatif FTP que vous avez l'habitude d'utiliser pour télécharger/téléverser des fichiers.

Avant de commencer, récupérer l'archive fournie sur Moodle. Le répertoire "emission" (respectivement "reception") dans le répertoire où sera lancé le programme client (respectivement serveur) est nécessaire. Vous pouvez placer ces nouveaux répertoires ailleurs, mais votre code sera à adapter). Dans le répertoire "emission", placez des fichiers de différentes tailles et différents types (au moins un de quelques centaines d'octets, un de quelques centaines de Ko et un de quelques centaines de Mo).

L'objectif est d'écrire :

— un programme client qui 1) se connecte à un serveur, 2) envoie un fichier se trouvant dans le répertoire "./emission", 3) affiche le nombre total d'octets effectivement envoyés depuis le début des communications avec le serveur ainsi que le nombre total d'appels de la fonction send(...), et 4) termine. L'adresse IP et le numéro de port du serveur ainsi que le nom du fichier à envoyer sont des paramètres de votre programme.

— un programme serveur itératif qui pour chaque client connecté : 1) reçoit un fichier et le stocke dans le répertoire "./reception" 3) affiche le nombre total d'octets effectivement reçus depuis le début des communications avec **ce client** ainsi que le nombre total d'appels de la fonction recv(...) (toujours avec ce client seulement) et 4) termine proprement la connexion avec le client avant de passer au suivant. Le numéro de port de la socket d'écoute des demandes de connexions est à passer en paramètre de votre programme.

Avant de programmer, il est nécessaire de réfléchir à ce qui suit :

1. Pour envoyer (respectivement recevoir) le contenu d'un fichier, il est nécessaire de savoir lire un contenu depuis un fichier (respectivement, écrire un contenu dans un fichier). Ceci est supposé être acquis. Toutefois, pour vous aider, des bouts de code sont fournis. Lire et comprendre ce code avant de poursuivre. La lecture et l'écriture dans un fichier se fera par blocs d'octets et non en une seule fois. Discutez en avec votre chargé(e) de TP.

2. Pour envoyer et recevoir un fichier, il et nécessaire de définir un protocole d'échanges entre le client et le serveur. Ce protocole doit permettre l'envoi correct du nom de fichier et de son contenu. Définir un tel protocole en prenant bien en compte la gestion des limites des messages.

3. En utilisant deux machines différentes, tester votre application sur différents exemples de fichiers. Le fichier envoyé et le fichier reçu sont-ils identiques ? Si ce n'est pas le cas, essayer de trouver l'erreur et demander de l'aide à votre encadrant(e).

Aide

Pour savoir si deux fichiers sont identiques, utiliser la commande "diff". Exemple : l'exécution de "diff fichier1 fichier2", affichera la différence de contenu entre fichier1 et fichier2. Si les fichiers sont identiques, rien ne s'affiche sur la sortie standard. Pour votre exercice, les fichiers étant sur des machines différentes, il sera bien-sûr nécessaire de les mettre sur la même machine pour les comparer. Plusieurs solutions sont possibles : la commande scp, l'envoi par email, ou autre, si vous avez travaillé en binôme (pas pour des fichiers volumineux), ou en renvoyant au client le fichier reçu par le serveur (ce qui pousserait plus loin le test de vos programmes).

Pour la manipulation d'un fichier, les fonctions suivantes sont utilisées dans le code fourni. Vous êtes invités à lire la documentation pour plus de détails. Vous pouvez choisir d'autres fonctions à partir du moment où vous maîtrisez bien ce que vous faites.

— pour l'ouverture d'un fichier :

FILE * fopen(const char *restrict filename, const char *restrict mode) ;

— pour connaître la taille d'un fichier :

int stat(const char *path, struct stat *buf) ;

— pour lire dans un fichier :

size_t fread(void *restrict ptr, size_t size, size_t nitems, FILE *restrict stream) ;

— pour écrire dans un fichier :

size_t fwrite(const void *restrict ptr, size_t size, size_t nitems, FILE *restrict stream) ;

— pour fermer un fichier :

int fclose(FILE *stream) ;

En bonus (optionnel pour les plus curieux) pour connaître la taille du buffer de réception ou d'envoi d'une socket, vous pouvez :

— soit consulter le contenu de : /proc/sys/net/ipv4/tcp_rmem
— soit utiliser la fonction
int getsockopt(int socket, int level, int option_name, void *restrict option_value, socklen_t *restrict option_len) ;
en passant en paramètre SOL_SOCKET pour level, SO_SNDBUF (ou SO_RCVBUF) pour option_name, l'adresse d'un entier pour option_value et la taille d'un entier pour option_len. option_value permet de récupérer la taille du buffer d'envoi (ou de réception).
— soit une autre solution de votre choix.

2 Transfert de fichiers : traitement de plusieurs clients

Dans l'exercice précédent, le serveur est itératif : il traite un client après l'autre. Si vous lancez un premier client qui transfert un fichier de taille importante (quelques Go) et que vous lanciez d'autres clients en parallèle (pour transférer aussi des fichiers volumineux), que se passe t-il pour ces nouveaux clients ?

Vous avez compris qu'un serveur itératif peut ne pas être adapté dans certains cas. Nous souhaitons donc modifier le serveur pour qu'il puisse recevoir, en même temps, des fichiers en provenance de différents clients.

1. Modifier le serveur pour traiter en parallèle les clients. L'idée est de mettre en place un serveur concurrent multi-processus en utilisant la fonction fork(...). Remarque : il est nécessaire de repenser le stockage des fichiers pour qu'il n'y est pas de conflit entre deux clients qui envoient des fichiers portant le même nom : il y a conflit si ces deux fichiers sont stockés dans le même répertoire coté serveur ("./reception" dans l'exercice précédent). Le serveur peut par exemple créer, pour chaque client, un sous-répertoire dédié dans "./reception"
2. Tester votre application avec plusieurs clients.