Clément DIF

Sylvain BESSONNEAU

Compte rendu du Projet en C
Programmation et système

I. Présentation sommaire du projet

Nous avons souhaité créer un logiciel mêlant à la fois un système de transfert de fichiers et un chat en réseau local en multithreading synchronisé. Notre projet regroupe d'ailleurs deux exécutables distincts : un pour le serveur et un pour les clients. La mise en place du chat s'effectue par l'ouverture, en premier lieu d'un serveur sur un port précis et par la connexion ultérieure de plusieurs clients via l'adresse IP fournie par le serveur (*localhost* si le serveur et le client sont s'exécutent sur la même machine ou obtenue grâce à *ifconfig*).

Le transfert de fichiers est une fonctionnalité complémentaire puisqu'il est réalisable en pleine session de chat. En effet, dans le cas de la saisie par l'utilisateur ou de la réception d'une requête « /sendto »*, le transfert se lance par le biais d'un thread totalement indépendant. Il est alors possible de faire transiter un fichier en texte ASCII ou tout simplement binaire à un autre utilisateur déterminé qui aura au préalable donné son accord.

* : Cette dernière sera expliquée plus en détail en II.B.a

II. Guide d'utilisation rapide

Notre application est simple d'utilisation. Les deux exécutables (client et serveur) peuvent être compilés à l'aide d'un Makefile fourni. Il est cependant nécessaire d'installer au préalable la bibliothèque « libncurses » à l'aide de la commande sudo apt install libncurses5-dev sur une distribution Debian ou dérivée comme Ubuntu.

A. Le serveur

L'exécutable prend seulement un paramètre, il s'agit du port de la socket passive. Un message indique par la suite le bon fonctionnement du serveur et l'aptitude d'accepter des nouveaux clients (10 au maximum). Il est possible d'interagir avec les clients en écrivant des messages sur le terminal et en validant avec la touche « Entrée ». Par ailleurs, l'on a la possibilité de rentrer des commandes. Celles-ci sont d'ailleurs listées ci-dessous :

- /list pour récupérer la liste des clients connectés
- /quit pour fermer le serveur
- /kick <username> pour déconnecter un utilisateur
- /abort pour forcer l'arrêt d'un transfert en cours

B. Le client

Avant toute chose, il est préférable d'exécuter toujours en plein écran pour disposer de l'interface la plus agréable possible et de la place nécessaire pour taper les messages et les commandes. En effet, le terminal ne pourra pas être redimensionné après le démarrage de l'interface graphique (voir la section IV. Remarques annexes).

L'exécutable prend trois paramètres : le pseudonyme souhaité par l'utilisateur, l'adresse IP du serveur et le port du serveur. Si les arguments sont intègres et corrects, l'interface graphique démarre. Elle est divisée en deux parties. La section supérieure correspond aux messages reçus et aux informations écrites par le programme et destinées à l'utilisateur. La partie du bas, quant à elle, permet à l'utilisateur de saisir du texte ou des commandes et de valider avec la touche « Entrée ». Les commandes disponibles sont listées ci-dessous :

- /list pour récupérer la liste des clients connectés
- /quit pour quitter le chat
- /sendto <dest_username> <dir>

La commande /sendto requiert un nom d'utilisateur en premier argument séparé par un espace et un chemin absolu de fichier en second argument également séparé par un espace.

Exemple : /sendto alice /home/bob/mon_fichier.txt

III. Description du fonctionnement du programme

A. Le chat

a) Côté client

- Phase d'initialisation

Pour compiler l'exécutable relatif au client, nous avons besoin du fichier *client.c* contenant le code source principal du client, du fichier *client_functions.c* contenant les définitions des différentes fonctions nécessaires au bon fonctionnement du script principal du client et des fichiers headers *client.h* et *client_functions.h* contenant les « *defines* », les définitions des structures et les déclarations des fonctions (prototypes). L'on aura évidemment pensé aux « *include guards* »

En premier lieu, il est judicieux de préciser que le programme attend des arguments lors de son exécution : le pseudonyme de l'utilisateur, l'adresse IP du serveur ainsi que le port utilisé par ce dernier.

Ensuite, après vérification de l'intégrité des arguments, le programme se connecte au serveur à l'aide de la fonction *connectSocket* (deux sockets seront utilisées : une pour les messages du chat et l'autre pour les fichiers). Par la suite, l'on procède à l'envoi du pseudonyme et l'on attend la réponse du serveur pour savoir si ce nom d'utilisateur est déjà pris par un autre client. A noter que le serveur souhaite recevoir un accusé de réception de sa réponse pour conserver une certaine synchronisation. En effet, dans la suite, le serveur envoie un message de bienvenue au client nouvellement connecté et il ne faudrait évidemment pas que ce message vienne écraser la confirmation ou l'infirmation de la non utilisation du pseudonyme.

Dans le cas où le client peut effectivement se connecter et rejoindre la session sans encombre, l'on initialise l'interface graphique avec la fonction *initscr*, l'on active la récupération de la saisie clavier avec la fonction *keypad* et l'on divise l'interface en deux parties (*top_win* et *bottom_win*) avec la fonction *subwin*. Ces deux divisions du terminal nous permettront de distinguer la saisie de l'utilisateur de la réception des messages.

Pour finir l'initialisation de l'interface graphique, l'on personnalise cette dernière et on l'affiche à l'écran avec la fonction *initInterface*. Cette fonction crée les cadres et insère du texte pour aider l'utilisateur (par exemple : « Message : »).

Par la suite, l'on crée un tableau pour conserver la conversation qui va contenir les messages envoyés et reçus. Le nombre et la taille des messages que peut contenir cette conversation dépend de la largeur et de la hauteur de la fenêtre.

- Boucle du programme

La boucle principale du programme écoute la socket message du client (*msg_server_sock*) ainsi que l'entrée standard pour détecter un éventuel changement d'état via la fonction select et effectuer les actions correspondantes. L'on distinguera alors deux cas.

1) Si la socket message du serveur est activée, on reçoit le message dans un buffer nommé *« buffer »* (pas très original…) via la fonction *recvServer* qui s'occupe d'ailleurs de tester la réception. Si le message commence par *« /sendto »* cela signifie qu'un autre utilisateur veut transférer un fichier. L'on passera alors en traitement de requête (voir partie B. Le transfert de fichiers). Sinon, on affiche simplement le message avec la fonction *writeInConv*. Cette dernière se charge d'ailleurs d'écrire le message au bon endroit dans la conversation. Si cette dernière se trouve remplie, la fonction procède alors à un décalage des messages vers le haut.

2) Si c'est le descripteur de fichier de l'entrée standard qui présente un changement, l'on récupère le caractère saisi avec la fonction *getch* et on le stocke dans un buffer nommé *« msg_buffer »*. Il est important de préciser que ce buffer est totalement indépendant de celui assigné à la réception des messages (voir cas précédent) pour éviter tout mélange de données puisque l'on procède à une lecture caractère par caractère. Avant de stocker ledit caractère, l'on vérifie si l'utilisateur a appuyé sur une touche spéciale. Si c'est « flèche de droite » ou « flèche de gauche », on décale le curseur. Si c'est la touche retour, on efface le dernier caractère.

Par ailleurs, si la touche entrée est saisie ou si l'utilisateur a rempli toute la ligne, alors on vérifie si le message entré est *« /quit »*. Si c'est effectivement le cas, l'on efface les éléments en mémoire avec la fonction *clearMemory* et l'on déconnecte l'utilisateur. (Si le message est *« /abort »* ou *« /sendto »*, se référer à la partie B consacrée au transfert de fichiers).

Si le message ne correspond à aucun des critères définis précédemment et s'il n'est pas vide, on l'envoie au serveur et on l'écrit dans la conversation, précédé du mot *« Vous : »*.

Enfin, l'on efface le message entré du cadre de saisie et l'on rafraîchit l'interface avec la fonction *convRefresh*.

A noter que l'on aurait pu faire la saisie du message avec la fonction *getnstr* de *ncurses* qui permet de récupérer un certain nombre de caractères. Cette fonction gère aussi les flèches et les effacements. Cependant, nous ne l'avons pas utilisée car dès que l'on rentrait un caractère, elle bloquait la réception des messages jusqu'à l'appui sur la touche entrée. En fait, on ne retournait pas au début de la boucle while lors de la saisie de chaque caractère.

b) Côté serveur

- Phase d'initialisation

Pour compiler l'exécutable du serveur, nous avons besoin du fichier *server.c* contenant le code source principal du serveur, du fichier *server_functions.c* contenant les définitions des différentes fonctions nécessaires au bon fonctionnement du script principal du serveur et des fichiers headers *server.h* et *server_functions.h* contenant les « *defines* », les définitions des structures et les déclarations des fonctions (prototypes).

Premièrement, le programme attend un numéro de port en paramètre (il s'agira du port pour la socket passive). Il s'en suit alors la création d'un tableau de structures *struct Client* (définie dans le header « *server_functions.h* ») pour conserver l'ensemble des clients connectés. La structure en question est composée de trois variables : un tableau de 16 *char* pour conserver le nom d'utilisateur du client et deux *int* représentant respectivement la socket relative aux messages et celle relative au transfert de fichiers pour le client concerné. L'on crée également une variable *clients_nb* qui contiendra le nombre de clients actuellement connectés ainsi que deux buffers nommés « *buffer* » et « *formatting_buffer* ». Le premier, que l'on qualifiera de « buffer standard » sert, en particulier, à recevoir les messages des clients et à envoyer les messages du serveur une fois formatés. En effet, les paquets transitant sur le réseau pour notre application étant toujours au maximum de 1 024 octets, dans le cas de la réception d'un message d'un client de 1 024 octets, afin d'éviter un « buffer overflow », il nous est nécessaire, si l'on veut accoler le pseudonyme devant ledit message (étape de formatage), de faire appel à un autre buffer de taille plus grande. On affectera alors le 1 024^{e} octet de ce buffer de formatage à '\0' avant de le copier dans le buffer standard pour finalement passer à l'envoi.

Enfin, pour terminer l'initialisation, le serveur démarre l'écoute à l'aide de la fonction *listenSocket*.

- Boucle du programme

La boucle principale du programme écoute la socket passive du serveur (*passive_server_sock*), les sockets message des clients (*msg_client_sock*) ainsi que l'entrée standard pour détecter, de la même manière que pour le client, un éventuel changement sur ces descripteurs de fichiers via la fonction select. L'on distinguera alors trois cas :

1) Si la socket passive du serveur a été activée, l'on ajoute le client avec la fonction *newClient* puis l'on vérifie que le pseudonyme du nouvel utilisateur n'est pas déjà pris par quelqu'un d'autre. Si c'est effectivement le cas, on envoie une réponse négative à sa tentative de connexion pour finalement le supprimer. Sinon, on lui envoie une réponse positive et l'on attend l'accusé de réception avant de lui souhaiter la bienvenue. Les autres clients n'en demeurent pas en reste car ils recevront également un message leur signalant qu'un nouveau client s'est connecté.

2) Si c'est la socket relative aux messages d'un des clients qui a été activée, alors on récupère ledit message avec la fonction *recvClient*. Dans le cas où cette fonction retourne 0, le client s'est déconnecté. On le signale alors aux autres, en leur envoyant un message avec la fonction *sendToOther* et l'on supprime formellement le client concerné avec la fonction *rmvClient*. Cette fonction réorganise d'ailleurs le tableau des structures *struct Client*.

Si le message reçu commence par « /sendto », alors on procède à un traitement de requête (voir la partie B dédiée au transfert de fichiers).

Si le message reçu est de type « /list », on liste les pseudonymes des personnes connectées et on envoie cette liste à l'utilisateur qui les a demandés via la fonction *sendClient*.

Si le message est de type « /abort », l'on met fin au transfert en cours (voir la partie B consacrée au transfert de fichiers).

Sinon, on envoie simplement le message reçu à tous les aux autres clients via la fonction *sendToOther*.

3) Si c'est le descripteur de fichier de l'entrée standard qui signale un changement, alors, on récupère le message avec la fonction *fgets*. On pensera d'ailleurs à vider le flux d'entrée standard pour éviter la présence de « résidus ».

Si le message est de type *« /abort »*, on arrête le transfert en cours (plus de détails dans la partie B consacrée au transfert de fichiers). Si le message est de type *« /quit »* alors on déconnecte le serveur. Si le message commence par *« /kick …»* on vérifie que le pseudo existe puis on déconnecte le client concerné. Sinon, si le message est *« /list »*, on affiche les pseudos des personnes connectés.

Enfin, si le message ne correspond à aucun des cas précédent, il s'agit simplement d'un message standard et on l'envoie à tous les clients à l'aide de la fonction *sendToAll*.

B. Le transfert de fichiers

a) Côté client

Nous allons d'abord nous intéresser à la partie relative à l'envoi du fichier. Pour expliquer de la manière la plus concise mais néanmoins précise il est utile de rappeler que, le client, lors de la session de chat, analyse toujours la saisie de l'utilisateur sur le flux « entrée standard ». Dans le cas d'une commande (commençant par un slash), le programme va s'intéresser au sens de cette commande pour la traiter.

En particulier, il est possible que l'utilisateur ait entré une requête du type *« /abort »*. Cependant, cette commande n'est autorisée que pour le serveur ou lors d'une erreur dans le transfert. Un message est alors affiché au client lui indiquant que cette commande n'est pas autorisée.

Néanmoins, il est également possible que l'utilisateur ait saisi une requête de type « /sendto <dest_username> <dir> ». Dans ce cas, l'on va d'abord regarder s'il est possible de verrouiller le mutex associé à la variable « *thread_status* » représentant l'état du thread relatif à l'envoi et à la réception d'un fichier.

Dans le cas d'un verrouillage effectif, nous avons accès à la variable *thread_status* en écriture et en lecture et il nous est alors possible de vérifier la valeur de cette dernière. Après vérifications, l'on peut déterminer si un thread de transfert est déjà en cours ou non. Dans le premier cas, on signale tout simplement à l'utilisateur de retenter sa chance plus tard et dans le second, on passe à l'étape suivante.

Il s'agit alors de vérifier la contenance de la requête : on fait ainsi appel aux fonctions *verifySendingRequest* et *verifyDirectory* qui examinent respectivement la syntaxe de la requête et le chemin du fichier (notamment son existence). Si les vérifications échouent, on signale au client le problème et on revient au chat.

Dans le cas positif, l'on envoie au serveur la requête brute sur la socket des messages pour que celui-ci puisse la faire parvenir au destinataire. Suite à cela, l'on reçoit une réponse matérialisant l'un des cas suivants : le destinataire n'est pas connu par le serveur, un transfert est déjà en cours entre d'autres clients, le destinataire a refusé la requête ou enfin, le destinataire a accepté le transfert.

Evidemment, si la réponse n'est pas favorable, l'on revient au chat tout en prévenant l'utilisateur. Dans l'opportunité inverse, l'on se prépare à lancer le thread relatif au transfert à proprement parlé exécutant la fonction *transferSendControl*. Cela se prépare d'ailleurs par l'initialisation d'une structure de type *struct TransferDetails* car les fonctions exécutées par les threads ne prennent toujours qu'un pointeur universel en argument (*void **). La structure est alors composée d'un pointeur sur le chemin d'envoi du fichier*, du socket pour les fichiers, du socket pour les messages, d'un pointeur vers le mutex, d'un pointeur sur int vers la variable *thread_status* qui est d'ailleurs affectée juste avant le lancement du thread pour matérialiser l'état « en cours d'exécution », d'un double pointeur sur char pour avoir accès à la conversation, d'un pointeur sur int pour la ligne active de la conversation, d'un pointeur sur le cadre du haut et d'un pointeur sur le cadre du bas de l'interface graphique.

A noter que, juste après la création du thread, l'on pense évidemment à déverrouiller le mutex qui sera par la suite verrouillé par le thread en question.

* : le chemin est un pointeur vers un caractère et non un tableau de caractères car l'allocation a déjà été effectuée dans le thread principal et le tableau ne sera pas modifié tant que le thread du transfert n'est pas terminé.

Dans le thread, les erreurs rencontrées engendreront l'arrêt de ce dernier ainsi que l'envoi d'un message de type *« /abort »* au serveur pour que ce dernier signale au client destinataire qu'il y a eu un problème. Vis-à-vis du transfert, l'on commence d'abord par retenter l'ouverture du fichier car il est possible que ce dernier ait été, entre temps, supprimé ou modifié. Par la suite, l'on envoie au serveur le nombre de paquets de 1 024 octets que l'on va livrer ainsi que la taille du dernier paquet (0 si le fichier a une taille qui est un multiple de 1 024). Enfin, on passe à l'envoi des dits paquets avec l'attente à chaque fois d'un accusé de réception pour garder une synchronisation. En effet, en l'absence de cette dernière, il est possible que l'émetteur envoie plus vite que le destinataire reçoit ce qui se traduirait par la réception d'un fichier au final incomplet voire corrompu.

En fin de transfert, l'on ferme le thread en songeant nécessairement à modifier la variable « *thread_status* » en conséquence et à évidemment retirer le verrou sur le mutex. Le client est d'ailleurs notifié du bon déroulement du transfert.

Concernant la réception, l'approche est très similaire, quasi symétrique, c'est pourquoi les explications de cette section seront plus brèves.

De la même manière que pour l'entrée standard, la socket relative aux messages est surveillée en permanence chez le client. L'on peut alors recevoir un message quelconque qui sera affiché à l'écran mais il est également possible de recevoir une commande. Celle-ci peut-être du type *« /abort »* (énoncé précédemment) demandant de mettre fin au thread relatif au transfert en cours et entraînant l'arrêt total du programme ou du type *« /sendto ».* On vérifie alors, de la même manière qu'énoncée précédemment, l'état du thread du transfert via *« thread_status »* et l'on passe alors en traitement de requête si c'est possible. L'utilisateur est effectivement interrogé via la fonction *answerSendingRequest* pour savoir s'il souhaite recevoir le fichier dans un dossier nommé « File_Transfer » créé à l'avenir dans son répertoire personnel (ou home directory dans la langue de Shakespeare). Cette réponse est alors transmise au serveur sur la socket des fichiers (explications du pourquoi ce socket dans la partie b. relative au serveur).

A noter que si, un fichier du même nom existe déjà dans le dossier File_Transfer, la réception est d'office refusée accompagné d'un message relatant la situation.

La structure passée (par adresse) en argument au thread qui appellera la fonction *transferRecvControl* est la même que lorsque l'on procède à un envoi à ceci près que le chemin est évidemment non pas celui du fichier à expédier mais bien celui du fichier dans lequel écrire.

Dans le thread du transfert, l'on reçoit le nombre de paquets et la taille du dernier par le serveur qui ne sert en fait simplement que de médiateur. L'on passe alors à la réception à proprement parlé tout en envoyant un accusé de réception suite à chaque paquet reçu.

En fin de transfert, l'on ferme également le thread en songeant à modifier la variable *« thread_status »* en conséquence et à évidemment retirer le verrou sur le mutex. Le client est d'ailleurs notifié du bon déroulement du transfert.

b) Côté serveur

Comme expliqué dans la partie A, le serveur est toujours à l'écoute des sockets relatives aux messages des clients. Il est donc possible, de recevoir une requête de type « /sendto ». Dans ce cas effectif, l'on procède, comme chez le client, à une tentative de verrouillage du mutex associé à la variable « *thread_status* », si le verrouillage échoue ou si la variable « *thread_status* » contient une valeur matérialisant l'existence d'un thread de transfert en cours, une réponse négative est alors envoyée au client émetteur et une trace de cet évènement est affiché sur le terminal du serveur.

Dans le cas où un thread n'est pas déjà en cours, l'on vérifie alors l'existence du pseudonyme du client destinataire. S'il n'existe pas de client connecté correspondant, une réponse négative est envoyée au client.

Dans le cas contraire, l'on se prépare à lancer le thread du côté serveur appelant la fonction *transferControl*. Dans cet objectif, l'on initialise une structure de type « *struct TransferDetails »* qui sera passée par la suite par adresse au thread. Cette dernière est constituée de deux structures *struct Client* représentant respectivement le client émetteur et le client destinataire, d'un pointeur sur un char nommé request pointant vers la requête « /sendto » brute*, d'un pointeur sur int vers la variable thread_status qui est d'ailleurs affectée juste avant le lancement du thread pour matérialiser l'état « en cours d'exécution » et enfin d'un pointeur vers le mutex.

A noter que, juste après la création du thread, l'on pense évidemment à déverrouiller le mutex qui sera par la suite verrouillé par le thread en question.

* : La requête est un pointeur sur un char et non un tableau de char car cette dernière a déjà été allouée dans le thread principal et l'on est sûr qu'elle ne sera pas modifiée tant que le transfert n'est pas terminé.

Au début de l'exécution du thread relatif au transfert, l'on envoie, sur la socket des messages, la requête au destinataire pour savoir si ce dernier souhaite effectivement recevoir le fichier et l'on reçoit sa réponse sur la socket des fichiers. Il est important d'expliquer un tel choix au niveau de la réception de la réponse : en effet, si nous avions entrepris de recevoir cette dernière sur la socket des messages, cela serait totalement entré en conflit avec le thread principal qui écoute également sur la socket des messages de chaque client.

Concernant la réponse du client destinataire, celle-ci est analysée puis envoyée au client émetteur. Dans le cas d'un refus, une trace est affichée sur le terminal du serveur et l'on quitte

le thread en pensant à changer la valeur de « *thread_status* » et à relâcher le verrou sur le mutex. Dans le cas contraire l'on passe à l'étape suivante décrite au paragraphe suivant.

Par la suite, l'on reçoit le nombre de paquets et la taille du dernier paquet qui vont être envoyés. L'on pense à les stocker en mémoire et à les transmettre au client destinataire. Par ailleurs, ce dernier va alors nous envoyer un accusé de réception que l'on transmettra au client émetteur.

Le transfert, à proprement parlé, peut alors commencer, l'on va pour chaque paquet, le recevoir, le transmettre au destinataire et récupérer l'accusé de réception de ce dernier pour finalement le renvoyer à l'émetteur.

Suite à cette étape, l'on affiche un message sur le terminal manifestant le bon déroulement du transfert, l'on affecte « *thread_status »* à la valeur correspondant à l'état « en attente de lecture du code de retour du thread » et l'on libère le verrou juste avant de quitter le thread.

A noter qu'il est également possible de recevoir une requête de type « /abort ». Cette dernière ne pouvant être envoyée que lors d'une erreur pendant le transfert, l'on est sûr qu'un transfert était bien en cours si l'on en reçoit effectivement une. A l'aide de la structure ayant servi au thread et n'ayant pas été modifiée car il ne peut y avoir deux transfert simultanés, l'on fait parvenir la requête « /abort » à l'autre interlocuteur pour que ce dernier soit informé de l'annulation du transfert.

Par ailleurs, il est probable que la commande « /abort » ait été rentrée sur l'entrée standard. L'on procède alors à une tentative de verrouillage du mutex et de lecture de la variable *thread_status*. Si ces deux vérifications indiquent qu'un transfert est effectivement en cours, on envoie cette commande aux deux clients concernés (obtenus via la structure de type *struct TransferDetails*) et l'on déverrouille le mutex, sinon, l'on affiche simplement sur le terminal qu'aucun transfert est en cours.

IV. Remarques annexes

- Si l'utilisateur veut envoyer un fichier qui est dans un dossier dont le chemin ne tient pas sur la ligne d'écriture, il faut que l'utilisateur ouvre la fenêtre en plein écran (cela passe par un redémarrage de l'application).

- Pour des fichiers nécessitant un transit de plus de 500 000 paquets, (donc de taille supérieure à 512 Mo), le programme affiche de temps en temps le pourcentage d'avancement du transfert chez le serveur mais également chez chaque client.

- Le programme, autant du côté serveur que du côté client, risque d'afficher des messages d'erreur ou simplement d'information. Ils sont balisés de la façon suivante : < FTS > précède un message de type informatif et relatif au transfert de fichiers (d'où l'acronyme File Transfer Service), < ERROR > précède une erreur qui n'a pas entraîné un arrêt du programme et < FERROR > indique une erreur fatale.

- Il est possible que l'interface graphique de l'application ne supporte pas une tentative de redimensionnement ou un changement de focus. En effet, notamment sous Unity, ncurses n'apprécie pas les animations lorsque l'on change de focus sur une autre fenêtre. Il est ainsi préférable d'utiliser la combinaison ALT+TAB pour changer de focus si l'on souhaite notamment travailler en « localhost ».

- Dans le code source du programme, les variables sont toujours écrites en snake_case, les fonctions en camelCase et les structures en PascalCase.

- Remarquez que l'on n'a pas utilisé de « typedef » et pourtant on en avait l'envie…