Clément DIF

Sylvain BESSONNEAU

Compte rendu du Projet en C

Programmation et système

1. 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. 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 (*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

1. Description du fonctionnement du programme
2. Le chat
3. Côté client

Pour faire fonctionner l’exécutable client, nous avons besoin du fichier client.c contenant le code source du client, du fichier client\_functions.c contenant les différentes fonctions du client et des fichiers headers client.h et client\_functions.h.

Tout d’abord, le programme attend des arguments en paramètres : le pseudo de l’utilisateur, l’adresse IP du serveur ainsi que le port.

Ensuite, si le nombre d’arguments est correct, le programme se connecte au serveur avec la fonction connectSocket. Puis, lui envoie le pseudo de l’utilisateur et attend la réponse du serveur pour savoir si le pseudo existe déjà.

Puis, on initialise l’interface graphique avec initscr, on active la récupération de la saisie clavier avec la fonction keypad, on divise l’interface en deux parties (top\_win et bottom\_win) avec la fonction subwin.

Puis, on personnalise l’interface et on l’affiche avec la fonction initInterface. Cette fonction crée les cadres et met le texte (par exemple : « Message : »).

Ensuite, on crée le tableau de conversation qui va contenir les messages envoyés et reçus. La taille que peut contenir cette conversation dépend de la largeur et de la hauteur de la fenêtre. On fait l’initialisation à ce moment-là du programme car il faut que ce soit après la fonction initscr() car on a besoin de LINES pour définir un nombre de ligne dans la conversation.

Après, on rentre dans la bouche while(TRUE) qui écoute la socket message du client (msg\_server\_sock) ainsi que l’entrée standard.

Si la socket message du serveur est active, on reçoit le message. Si le message est « /sendto » c’est que un utilisateur veut transférer un message. (cf : B transfert de fichier). Sinon, on affiche le message avec la fonction writeInConv. Cette fonction se charge d’écrire le message au bon endroit dans la conversation et si la conversation est remplie, la fonction fait un décalement des messages vers le haut.

Si c’est l’entrée standard qui est active, alors on récupère le caractère saisie avec la fonction getch et on le stocke dans un buffer (msg\_buffer). Avant de stocker le caractère, 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.

Si la touche entrée est saisie ou si l’utilisateur a remplie toute la ligne, alors on vérifie si le message entré est « /quit ». Si c’est le cas, on efface les éléments en mémoire avec la fonction clearMemory et on déconnecte l’utilisateur. (Si le message est « /abort » ou « /sendto » se référer au B transfert de fichier).

De plus, le programme n’envoit pas le message si celui-ci est vide.

S’il n’est pas vide, on envoit le message au serveur et on écrit le message dans la conversation avec le mot « Vous ».

Et enfin, on efface le message entré et 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 utiliser car dès que l’on rentre un caractère, cette fonction bloque la réception des messages jusqu’à l’appui sur la touche entrée.

1. Côté serveur

Pour faire fonctionner l’exécutable serveur, nous avons besoin du fichier server.c contenant le code source du serveur, du fichier server\_functions.c contenant les différentes fonctions du serveur et des fichiers headers server.h et server\_functions.h.

Tout d’abord, le programme attend un numéro de port en paramètre. Puis, il ouvre le serveur avec la fonction listenSocket.

On créer un tableau de structure qui va permettre de stocker les clients qui sont connectés.

Après, on rentre dans la bouche while(TRUE) qui écoute la passive du serveur (passive\_server\_sock), les sockets message des clients (msg\_client\_sock) ainsi que l’entrée standard.

Si c’est la socket passive serveur qui a été activé, alors on ajoute le clients

Si c’est l’entrée standard qui a été activée, alors, on récupère le message entré avec fgets(). Puis, on ajoute « Serveur : » devant le message pour dire qui a envoyé le message. Ceci est fait à l’aide de la fonction concatener. Et enfin, on envoi le message à tous les clients à l’aide de la fonction envoiMessageTous(). Celle-ci utilise la fonction envoiMessage() dans une boucle for pour envoyer le message à tous les clients.

Si c’est un socket client qui a été activée, alors on lance la fonction ecouteMessage (). Cette fonction récupère le message du client ainsi que la taille. Si la taille est égale à -1 alors on ferme le serveur. Si la taille est de 0 alors cela veut dire que le client s’est déconnecté. Donc on envoi un message aux autres clients pour les informer que le client s’est déconnecté à l’aide de la fonction envoiMessageAutresClients(). Puis, on lance la fonction deconnexionClient(). Cette fonction va fermer le socket du client à l’aide de la fonction close() et supprimer le pseudo du client. Et elle va refaire la liste des socket clients et des pseudos. C’est-à-dire que la fonction va décaler le socket du client déconnecté à la fin de la liste. Pareil pour la liste des pseudos. Et on décrémente de 1 le nombre de clients.

Si la taille est différente de -1 et de 0, alors on concatène le pseudo au message, on affiche le message et on envoi le message aux autres client avec la fonction envoiMessageAutresClients().

Commandes :

Coté client : /list pour récupérer la liste des clients connecté

/quit pour quitter le chat

Coté serveur : /list pour récupérer la liste des clients connecté

/quit pour fermer le serveur

/kick … pour deconnecté quelqu’un

1. Le transfert de fichiers
2. 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 préciser 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 /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 vérifient 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 le 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 éxécutant la fonction transferSendControl. Cela se prépare d’ailleurs par l’initialisation d’une structure struct TransferDetails data car les fonctions éxécutées 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 et d’un pointeur vers la variable thread\_status qui est d’ailleurs affectée juste avant le lancement du thread pour matérialiser l’état « en cours d’exécution ». A noter que, juste après le thread, l’on pense évidemment à déverouiller 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 est 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 à modifier la variable « thread\_status » en conséquence et à évidemment retirer le verrou sur le mutex.

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, le socket relatif aux messages est surveillé 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 précedemment) demandant de mettre fin au thread relatif au transfert en cours ou du type « /sendto ». On vérifie alors de la même manière qu’énoncé 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 le 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 en argument au thread est la même que lorsque l’on envoie 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.

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.