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

Au niveau du client :

Il y a 2 fichiers : un fichier client.c et un fichier client.h.

Dans le fichier header (client.h) se trouvent les prototypes des fonctions ainsi que les #include et #define.

TAILLE \_BUF désigne la taille du buffer texte.

TAILLE\_PSEUDO désigne la taille maximale d’un pseudo soit de 15 caractères.

PORT signifie le port utilisé soit le port 54888 dans notre fichier.

Ensuite, le fichier client.c contient le code du programme du client.

Dans la fonction main :

On demande tout d’abord un pseudo avec la fonction demandePseudo(). Cette fonction vérifie que le pseudo ne dépasse pas 15 caractères.

Ensuite, on demande l’adresse IP pour se connecter au serveur avec la fonction demandeIP(). Cette fonction vérifie aussi que la taille de l’adresse IP est assez longue mais pas trop non plus.

Si le pseudo et l’adresse IP sont bons, alors on se connecte au serveur avec la fonction connect\_socket(). Cette fonction crée un socket, récupère l’adresse IP donnée par l’utilisateur et se connecte au serveur. Si ça n’a pas fonctionné, le programme ferme. Sinon la fonction retourne le socket.

Ensuite, on envoi le pseudo au serveur avec la fonction write\_serveur(). Cette fonction envoi un message au serveur grâce à la fonction send(). Si cela n’a pas fonctionné, on ferme le programme avec affichage d’un message d’erreur.

Si cela fonctionne, on libère la mémoire du pseudo et du mot de passe.

On initialise l’interface graphique avec initscr() puis on créer les deux parties de l’interface (fenHaut et fenBas) avec la fonction subwin().

Puis, on personnalise l’interface et on l’affiche avec la fonction initInterface(). Celle-ci va créer les cadres et mettre 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(1). Cette boucle infinie va permettre de rester continuellement à l’écoute du serveur et de l’entrée standard.

Au début de cette boucle, on se place au bon endroit dans la fenêtre pour pourvoi écrire des messages dans la fenêtre du bas. Cela se fait avec la fonction move().

Ensuite, on initialise le readfds à l’ensemble vide avec FD\_ZERO().

Puis on ajoute l’entrée standard et le socket au readfds avec la fonction FD\_SET().

On détecte une activité avec la fonction select() et on teste quel descripteur a été activé avec la fonction FD\_ISSET(). Si select() renvoi -1 alors on ferme l’interface, on lance la fonction effaceMemoire() et on quitte le programme avec un message d’erreur. La fonction effaceMemoire() efface les fenêtre, efface l’interface et ferme le socket.

Si c’est le socket qui a été activé, on lance la fonction read\_serveur(). Cette fonction récupère le message envoyé par le serveur avec la fonction recv() ainsi que la taille du message. Si la taille est -1 on lance la fonction effaceMemoire() et on arrête le programme. Si la taille est 0 alors on lance la fonction effaceMemoire() et on quitte le programme car le serveur s’est déconnecté.

Sinon, on vérifie que la taille reçue est inférieure à la largeur de la fenêtre.

Si ce n’est pas le cas, alors on met « … » a la fin du message.

Puis, on écrit le message reçu dans la conversation avec la fonction ecritDansConv().

Cette fonction vérifie si la conversation est remplie au maximum en nombre de ligne.

Si ce n’est pas le cas, alors on ajoute le message en dessous.

Si c’est le cas, alors on efface la conversation. On décale les messages vers le haut et on écrit le message reçu en bas de la conversation et on rafraichit l’interface avec la fonction rafraichit() qui efface la ligne d’écriture de la fenêtre du bas si jamais on a écrit du texte avec wclrtoeol(), qui redessine les cadres des fenêtres avec box() et qui rafraichit avec wrefresh().

Si c’est l’entrée standard qui a été activée, on récupère le message avec la fonction getnstr(). Cette fonction stoppe la saisie du message si la ligne est remplie.

Ensuite, on vérifie si le message est « déconnexion ». Si oui, alors on lance la fonction effaceMemoire() et on quitte le programme après avoir afficher déconnexion réussie.

Si ce n’est pas le message de déconnexion, alors on envoi le message au serveur avec la fonction write\_serveur(). Cette fonction envoie un message au serveur avec la fonction send() en vérifiant que l’envoi ai fonctionné. Puis, on concatène devant le message le « vous : », pour que cela fasse plus propre. Ceci est fait avec la fonction concatener(). Et enfin, on écrit ce message dans la conversation.

Au niveau du serveur :

Il y a aussi 2 fichiers : un fichier header serveur.c et un fichier serveur.c

Le fichier serveur.h contient le prototype des fonctions ainsi que les #include et #define

Le #define NB\_CLIENT\_MAX permet de définir le nombre de client maximal autorisé

Ensuite, le fichier serveur.c contient le code du programme du serveur.

Dans la fonction main :

Tout d’abord, on crée un socket, un tableau contenant les sockets des clients et un tableau contenant les pseudos des clients.

Ensuite, on ouvre le serveur avec la fonction ouvertureServeur(). Cette fonction utilise la fonction bind() ainsi que listen() et vérifie les erreurs au niveau de ces fonctions.

Après, on rentre dans la bouche while(1). Cette boucle infinie va permettre de rester continuellement à l’écoute du socket serveur, des sockets clients et de l’entrée standard.

Au début, on initialise le readfds à l’ensemble vide avec FD\_ZERO().

Puis on ajoute l’entrée standard, le socket serveur et les sockets clients au readfds avec la fonction FD\_SET().

On détecte une activité avec la fonction select() et on teste quel descripteur a été activé avec la fonction FD\_ISSET().

Si c’est le socket serveur qui a été activé, alors on ajoute le nouveau socket client à la liste des sockets (listesock) avec la fonction accept().

Puis, on lance la fonction ecouteConnexion(). Cette fonction reçoit le pseudo du client et vérifie que la taille reçue est différente de -1 et stocke le pseudo reçu dans la liste des clients (au même indice que le socket). Et enfin, cette fonction envoie un message de bienvenue au client avec la fonction envoiMessage(). Cette fonction envoiMessage() envoi un message au client grâce à la fonction send().

Puis, on envoie un message aux autres clients pour leur dire qu’un nouveau client s’est connecté. On le fait avec la fonction envoiMessageAutresClients(). Cette fonction utilise la fonction envoiMessage() dans une boucle for pour envoyer le message aux autres clients sauf au client qui vient de se connecter.

Et enfin, on augmente le nombre de clients de 1.

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.