Programmation reseaux : les sockets

Introduction aux reseaux

17 octobre 2025

Auteurs :

Paul Latu-Ferrand
Hugo BERTHOD

Encadrants :

O. Villain

Sommaire

1 Introduction

L'objectif de ce bloc de travaux pratiques est de réaliser un projet de calculatrice en réseau. Pour cela nous avons été amené à créer deux programmes, un serveur et un client suivant le protocole TCP. Nous avons donc choisi de travailler chacun sur un script différent, Paul sur le client et Hugo sur le serveur.

L'idée est que le client récupère les instructions via le terminal de l'ordinateur et les envoie au serveur qui traite le calcul et renvoie la réponse au client en l'affichant dans le terminal. Cette structure nécessite que certain paramètres soit commun au serveur et aux clients, par exemple la variable globale $BUFF\_SIZE$ qui fixe dans les deux programmes la taille maximale des chaînes de caractères envoyées, nous choisissons une taille max de 256. Nous avons utiliser la structure TCP suivante (qui est celle que le cours nous donne) :

Client TCP	Serveur TCP
`socket()`	`socket()`
	`bind()`
	`listen()`
`connect()`	**`accept()`**
`read()` **`write()`**	**`read()`** **`write()`**
`close()`	`close()`

FIGURE 1 – Structure TCP client serveur

Nous avons choisi pour travailler en collaboration d'ouvrir un repository sur github et de séparer en deux branches nos travaux afin de ne pas se gêner mutuellement.

2 Partie serveur

2.1 Serveur TCP

Le serveur est la partie du projet où se déroulent les calculs, nous avons choisi que le client envoie uniquement l'expression que l'on souhaite calculer. Mais avant de réaliser ce calcul, il faut mettre en place le serveur. Pour cela nous suivons simplement le déroulé suivant : création du socket, du bind et attente d'un client via *listen*. Une fois qu'un client s'est connecté, on traite son calcul en vérifiant qu'il respecte bien les règles mathématiques et on lui renvoie le résultat avant de fermer la connexion.

La partie création du serveur est reprise d'un exemple du cours et adaptée afin de convenir à notre projet. Toute la partie gérant le serveur TCP est contenu dans la fonction *main* qui prends en argument le numéro du port sur lequel on souhaite ouvrir notre serveur. Après avoir déclaré les différentes variables dont on va avoir besoin :

- $char\ buff[BUFF\_SIZE]$: buffer qui va nous servir pour sauvegarder les messages envoyés par les clients.
- $int\ i$: qui est le compteur du nombre de connexion que l'on a eu
- $int\ n$: qui va stocker le retour de $read$
- $struct\ sockaddr\_in\ sa$: qui est la structure de la socket du serveur
- $struct\ sockaddr\_in\ newsa$: qui est la structure de la socket pour les clients

On vérifie ensuite que nous avons bien 2 arguments en entrée du programme. Si il n'y en a qu'un seul c'est que le numéro du port souhaité a été oublié, s'il y a plus que deux arguments c'est qu'on a transmis trop d'informations lors de l'appel de l'exécutable ($argc$ permet de connaître le nombre d'arguments). On récupère ensuite le numéro du port que l'on assigne à la variable *port* et on initialise le socket *sa* de type $sockaddr\_in$. Si il n'y a pas d'erreur on crée le socket et on bind sur la socket. Une fois ces deux étapes effectuées, on initialise la file d'attente grâce à *listen*.

L'initialisation du serveur est terminée il faut maintenant se pencher sur la discussion avec le ou les futurs clients.

Lorsqu'un client tente de se connecter, on vérifie que cela est possible grâce à *accept* puis on incrémente le compteur du nombre de connexion du serveur. On affiche ensuite les informations sur le client dans le terminal du serveur afin de vérifier la cohérence avec les informations du coté client et on se place en écoute d'un message grâce à $read(newsd, buff, BUFFSIZE)$; qui va permettre "d'écouter" sur le flux de la socket connectée et de le stocker dans $buff$. On affiche ensuite les informations dans le terminal afin de pouvoir vérifier lors des tests que cela correspond bien aux données du client. On alloue ensuite une variable res_calc qui va contenir le résultat de l'opération de la calculatrice grâce à $char*\ res\_calc\ =\ (char*)malloc(strlen(buff)*sizeof(char))$, on utilise *malloc* avec la taille du buffer afin de ne réserver que le bon espace en mémoire. On utilise à la fin de la connexion avec le client $free(resc\_calc)$ pour libérer la mémoire allouée ce qui évite un overflow si on a beaucoup de clients.

On assigne ensuite le résultat de la fonction calculatrice à res_calc et on envoie au client le résultat avec la ligne $write(newsd,\ res\_calc,\ strlen(res\_calc))$; qui permet d'envoyer à la socket $newsd$, la chaîne de caractères res_calc de taille $strlen(res\_calc)$.

2.2 Parallélisation

Une fois que le processus fonctionne pour 1 seul client, on s'attelle à gérer plusieurs clients en parallèle. Pour cela on garde le même programme que l'on va venir encapsuler dans un processus fils, grâce à la fonction $fork()$. Cela permet au programme d'exécuter plusieurs partie du programme en parallèle afin que les clients puissent se connecter en même temps. On a donc un processus fils par client qui essaie de se connecter

On sépare donc le partie dite père ou principal, qui traite la gestion des connections entrantes et du serveurs global, de la partie fils qui s'occupe du traitement de chaque client et donc dans notre cas du calcul. L'allocation et libération de la mémoire ainsi que l'appel à la fonction calculatrice se font donc au sein du processus fils. Une fois que l'on a renvoyé le résultat au client, on ferme la discussion avec lui et on termine le processus fils associé.

2.3 Calculatrice

Le programme de la calculatrice est repris d'un projet de première année que nous avons adapté en une fonction au lieu que cette dernière soit incluse dans le main. Voici son prototype $char* \ calculatrice(char \ cal[], int \ n)$, elle prend en argument la chaine de caractère *calc* ainsi que sa taille. On déclare ensuite toutes les variables nécessaire au bon fonctionnement de la calculatrice qui sont :

- les deux chaines de caractères locales s_a et s_b
- le char *op* qui contiendra l'opérateur les entiers *nb*
- m qui est la taille de s_a
- k qui est la taille de s_b
- j qui est l'indice de l'opérateur dans la chaine de caractère *cal*
- *etat* qui permet de traiter les cas d'erreur en lui assignant différentes valeurs
- les doubles a et b qui représentes les deux opérandes du calcul
- le double res qui stockera le resutlat du calcul
- le pointeur char *ret* alloué par $char* \ ret \ = \ (char*)malloc((n+1)*sizeof(char))$ qui correspond à la chaîne de caractère que l'on va retourner

On utilise une nouvelle fois malloc afin d'allouer une taille suffisante mais pas trop grande en mémoire.

Une boucle for permet ensuite de trouver la position de l'opérateur et des opérandes et assigner à s_a et s_b les chaînes de caractère correspondantes. Pour l'opérateur on assigne tout simplement le caractère du string à *op*.

Une fois cela fait, on convertit avec l'aide de $atof$ les chaînes s_a et s_b en flottant a et b et si on a bien *nb* qui vaut 3 (ie. que les deux opérandes et l'opération ont bien étés détectés) alors on réalise le calcule et on passe *etat* à 1 (seul le calcul d'une division est différente à cause du traitement de la division par 0 qui peut conduire à $etat \ = \ 2$).

Ensuite selon les valeurs de *etat* nous avons 3 cas :

- $etat \ == \ 0$ alors on n'a pas réussi à reconnaître l'opérateur, on retourne donc une erreur.
- $etat \ == \ 2$ alors on a tenté de diviser par 0, on retourne également une erreur.

— $etat == 1$ alors tout c'est bien passé on retourne le résultat du calcul.

Pour retourner la bonne caractère, on utilise la fonction $sprintf(ret, "\%.02f", res)$ qui affecte à *ret* la chaîne contenu dans "%.02f" en utilisant la même syntaxe que printf. L'envoie du résultat au client est gérer par la fonction main du programme et non la calculatrice.

3 Partie client

Dans la partie client, l'objectif est de récupérer le calcul à effectuer, de l'envoyer au serveur puis de recevoir la réponse dans la foulée dans le même terminal.

Pour créer le code client, nous avons introduit plusieurs variables qui serons utiles dans la suite :

— *myname* qui est un pointeur sur le nom du programme

— *serveur* qui est le nom du serveur utilisé

— *port* qui est le port utilisé pour communiquer avec le serveur.

Il faut ensuite récupérer ces informations qui sont rentrées par l'utilisateur dans le terminal. Le moyen le plus simple de faire cela est d'utiliser *argc* qui renvoie le nombre d'éléments de la ligne de commande et *argv* qui contient ces éléments sous la forme d'un tableau de chaînes de caractères. Ainsi on a :

— *myname = argv[0]*

— *serveur = argv[1]*

— *port = atoi(argv[2])*.

Si on a plus de trois éléments dans la ligne de commande, alors le programme renvoie une erreur, pour cela on fait une simple boucle *if* sur la valeur de *argc*.

On peut ensuite s'attaquer à la partie principale de ce projet, la structure du socket, on initialise d'abord la structure internet *sockaddr_in* et on initialise le numéro de port obtenu plus tôt (voir ligne 52 à 60 du code *snippet_clientTcp.c*)

Pour récupérer les calculs à effectuer, nous avons utilisé la ligne de code *read(0, buf, BUFFSIZE)* que nous avons associé à un entier pour le réutiliser plus tard. Afin de ne pas surcharger le serveur en attendant que l'utilisateur rentre son calcul dans la ligne de commande, nous avons placé cette étape avant la création de la socket client et la connexion avec le serveur. J'ai également ajouté une boucle *while(1)* qui permet au programme de tourner et donc d'effectuer plusieurs calculs et la condition d'arrêt se fait au moyen d'un *if* qui break la boucle si on écrit *"quit"* dans le terminal (comme demandé par l'énoncé).

Après ces étapes on peut créer la socket et se connecter au serveur via la fonction *connect*, on envoie alors la requête au sevreur, cette dernière est écrite dans le terminal du client afin d'être sur que l'envoi s'est effectué et que le calcul demandé est le bon. En utilisant la fonction *read* on lit la réponse renvoyée par le serveur et on l'écrit également dans le terminal. On peut ensuite fermer la socket et passer à une autre requête grâce à la boucle while comme expliqué plus tôt et on décide de quitter le serveur quand on le souhaite.

Comme demandé dans l'énoncé du projet, le serveur doit pouvoir répondre à plusieurs clients en même temps. Ce processus de parallélisation n'affecte pas le code client et concerne uniquement celui du serveur, je n'ai donc rien eu à faire pour cela.

4 Test et Validation

Pour tester notre serveur et nos codes, nous avons simplement utilisé les commandes indiquées dans l'énoncé en adaptant le port à chaque fois en fonction de ceux qui étaient disponibles à cet instant.

Pour commencer nos tests, nous avons d'abord essayer d'établir une communication entre le client et le serveur, en rentrant un mot ou une phrase on voulait simplement que le serveur renvoie la même chose. Une fois que cette étape a été réalisée on a pu se concentrer sur les fonctionnalités propres à la calculatrices, à savoir des calculs avec les bons arrondis et dans une certaine plage de valeur.

Nous avons également essayé d'utiliser la bonne taille mémoire, c'est à dire de n'utiliser *malloc* avec en argument la taille des données et non $BUFF\_SIZE$ comme on aurait pu choisir pour des raisons de facilités.

De même nous avons, pour tester la gestion de plusieurs clients en parallèle, ajouté une temporisation afin de simuler un traitement plus long par client. Notre serveur est bien capable de répondre en parallèle à différents clients sans les faire attendre.

5 Conclusion

Ce projet nous a permis de nous familiariser pour la première fois avec la manipulation des sockets et de la parallélisation. Nous avo,ns pu travaille chacun sur une partie différente, tout en mettant en commun ce qu'on a appris et fait. En ayant choisi d'utiliser github, nus avons pu nous familiariser un peu plus avec cet outil, tout en utilisant le systèmes de branches qui est très pratique pour des travaux de groupes.

Finalement, Le serveur est fonctionnel et est capable de communiquer avec une dizaine de client sans dépendances entre eux. Il effectue les calculs avec les bonnes approximations sans erreurs.

6 Bibliographie

Références

[Repository Github]