Projet Système Réseau

BOU SERHAL Jean, CABROL Camille, FRANCES Tom, GOURDON Jérémie

November 30, 2021

Contents

Introduction

Le but de ce projet est de créer une application réseau client/serveur TCP/IP simplifiée, permettant l'échange d'images entre le client et le serveur. Le client pourra choisir de déposer ou de récupérer des images sur le serveur. Ce serveur acceptera les fichiers images déposés sous conditions d'admission, et devra assurer le service pour plusieurs clients simultanément.

Nous avons découpé ce projet en plusieurs étapes :

1. Création du noyau client/serveur
2. Conception de l'application (architecture et protocole d'échange)
3. Conception et implémentation des fonctionnalités de base (transfert de fichiers, critères d'admissibilité...)
4. Intégration des fonctionnalités au noyau

Chapter 1

Conception du noyau

Dans les 2 sections suivantes, nous avons décidé de gérer toutes les erreurs des fonctions système (*socket()*, *connect()*, etc) en affichant **errno** via la commande **perror()**, en profitant du fait que ces fonctions le positionnent.

Au départ, nos variables de type **sockaddr_in** étaient déclarées en tant que pointeurs, mais pour simplifier l'écriture du code et ne pas avoir à utiliser de **malloc()**, nous avons préféré ne pas utiliser de pointeurs, et de transmettre l'adresse de ces variables (**&nom_variable**) aux méthodes les utilisant en paramètre.

Nous avons choisi d'attribuer le port numéro 6067 au serveur, après avoir vérifié que ce port était libre grâce à la commande **netstat -t** listant tous les ports TCP ouverts.

1.1 Client

Nous avons tout d'abord créé une socket afin de permettre la communication entre le client et le serveur, puis nous avons récupéré les informations du serveur en utilisant le nom de notre machine (commande **gethostbyname()**).
Dans un premier temps, nous avons préféré travailler sur une même machine, en utilisant le nom **localhost**, afin de faciliter nos tests. Dans un second temps, une fois la création du noyau terminée et fonctionnelle, nous sommes passés sur l'utilisation de notre programme sur 2 machines distinctes, l'une jouant le rôle du client, l'autre du serveur. Pour cela, nous avons récupéré le nom de la machine serveur via la commande **hostname**, qui remplace le nom **localhost** dans le code de *Client.c*.
Ensuite, à l'aide de la commande **connect()**, nous avons établi une connection entre le client et le serveur. Une fois cette connection établie, nous avons choisi d'envoyer un entier au serveur via la méthode **write()**, pour des raisons de facilité (taille, lecture, écriture...).
Enfin, en utilisant la méthode **read()**, le client récupère une réponse de la part du serveur.

1.2 Serveur

Nous avons tout d'abord créé une socket permettant au serveur d'écouter les messages de clients, puis attaché cette socket au port choisi précédemment (commande **bind()**) : une socket étant un descripteur de fichier, il faut lui assigner un port afin de lui indiquer où elle doit écouter.
Afin d'éviter une accumulation trop importante de connections en attente, nous avons décidé de limiter leur nombre à 5 (car nous avons décidé que c'était suffisant dans notre cas).
Nous avons géré l'acceptation de la connection du client au serveur dans une boucle **while(1)** (1 car tout le temps vrai, ce qui nous permet de gérer manuellement la mort du serveur étant donné que les fonctionnalités s'occupant de cet aspect ne sont pas encore implémentées). Dans cette boucle, le serveur lit le message envoyé par le client via une socket de service renvoyée par la fonction **accept()**.
Pour le test de cette fonction, nous avons rencontré une erreur : nous souhaitions afficher un message lors de la réussite du test, hors rien ne s'affichait sur la console. Nous nous sommes rendu compte qu'il manquait un "**\n**" à la fin du **printf()**, car sans celui-ci, il n'y avait pas de flush du

buffer sur la sortie standard, donc pas d'affichage. Afin de permettre au serveur de traiter plusieurs connections simultanées, ce dernier délègue chaque nouvelle connection à un processus fils (1 fils par connection), créé au sein d'un **switch(fork())** :

cas -1 erreur de création du processus, on arrête le programme (**exit(-1)**)

cas 0 comportement du fils : fermeture de la socket d'écoute (car le processus fils ne l'utilise jamais), réinitialisation du comportement du signal **SIGCHLD** au cas où ce processus doive lui-même créer des fils.

cas default comportement du père, qui ici ne fait rien. On pourrait vouloir le faire attendre la mort de son fils à l'aide d'un **wait(NULL)**, mais en faisant cela, on bloquerait le programme qui ne pourrait ainsi pas traiter plusieurs connections et requêtes simultanément.